Multi-level checkpointing and silent error detection for linear workflows

Anne Benoit, Aurélien Cavelan, Yves Robert, and Hongyang Sun

RESEARCH REPORT

N° 8952

September 2016

Project-Team ROMA

ISSN 0249-6399   ISRN INRIA/RR--8952--FR+ENG

Multi-level checkpointing and silent error detection for linear workflows

Anne Benoit[*†], Aurélien Cavelan[*†], Yves Robert[*†‡], and Hongyang Sun[*†]

Project-Team ROMA

Research Report n° 8952 — September 2016 — 33 pages

Abstract: We focus on High Performance Computing (HPC) workflows whose dependency graph forms a linear chain, and we extend single-level checkpointing in two important directions. Our first contribution targets silent errors, and combines in-memory checkpoints with both partial and guaranteed verifications. Our second contribution deals with multi-level checkpointing for fail-stop errors. We present sophisticated dynamic programming algorithms that return the optimal solution for each problem in polynomial time. We also show how to combine all these techniques and solve the problem with both fail-stop and silent errors. Simulation results demonstrate that these extensions lead to significantly improved performance compared to the standard single-level checkpointing algorithm.

Key-words: resilience, fail-stop errors, silent errors, multi-level checkpoint, verification, dynamic programming.

[*] École Normale Supérieure de Lyon
[†] INRIA, France
[‡] University of Tennessee Knoxville, USA

RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Checkpoint multi-niveaux et détection des erreurs silencieuses pour des graphes de tâches linéaires

Résumé : Les erreurs fatales et silencieuses ne peuvent plus être ignorées sur des plates-formes à grande échelle. Des techniques de résilience efficaces doivent accommoder les deux types d'erreurs. Une approche traditionnelle de checkpoint et points de reprise peut être utilisée, en rajoutant des vérifications afin de détecter les erreurs silencieuses. Une erreur fatale entraîne la perte de tout le contenu mémoire, d'où l'obligation de faire une sauvegarde sur un support fiable (typiquement un disque). Pour gérer plusieurs types d'erreurs fatales, nous utilisons une approche de checkpoint multi-niveau sur supports stables. Par contre, nous utilisons des checkpoints en mémoire pour les erreurs silencieuses, ce qui donne des surcoûts bien plus faibles. De plus, les détecteurs récents offrent des mécanismes de vérification partielle, qui sont moins coûteux que les vérifications garanties, mais qui ne détectent pas toutes les erreurs silencieuses. Nous montrons comment combiner toutes ces techniques pour des applications HPC dont le graphe de dépendances est une chaîne de tâches, et nous donnons plusieurs algorithmes de programmation dynamique qui renvoient la solution optimale en temps polynomial. Des simulations démontrent que l'utilisation combinée de checkpoint multi-niveaux et de vérifications améliore la performance.

Mots-clés : résilience, erreurs fatales, erreurs silencieuses, checkpoint multi-niveaux, vérification, programmation dynamique.

Note

This report is an extended version of INRIA RR-8794, October 2015.

1 Introduction

Resilience is one of the major challenges for extreme-scale computing [18, 19]. Checkpointing [21] is the de-facto standard approach to dealing with *fail-stop* errors, defined as fatal interruptions (such as hardware or power failures, resource crashes) that call the faulty node for a reboot or replacement. However, the traditional single-level checkpointing method suffers from significant overhead [29, 15, 41], and multi-level checkpointing protocols now represent the state-of-the-art [38, 7, 25]. These protocols allow for different levels of checkpoints to be set, each with a different overhead and recovery ability. Typically, each level corresponds to a specific error type, and is associated with a storage device that is resilient to that type. The main idea of multi-level checkpointing is that checkpoints are taken for each level of faults, but at different rates. Intuitively, the less frequent the faults, the longer the interval between two checkpoints: this is because the risk of a failure striking is lower when going to higher levels; hence the expected re-execution time is lower too; one can safely checkpoint less frequently, thereby reducing failure-free overhead (checkpointing is useless in the absence of a fault).

In this paper, we first consider a very general scenario, where the platform is subject to k levels of fail-stop errors, numbered from 1 to k. Level ℓ is associated with an error rate λ_ℓ, a checkpointing cost $C^{(\ell)}$, and a recovery cost $R^{(\ell)}$. A fault at level ℓ destroys all the checkpoints of lower levels (from 1 to $\ell - 1$ included) and implies a rollback to a checkpoint of level ℓ or higher. Similarly, a recovery of level ℓ will restore data from all lower levels. As mentioned, fault rates are decreasing and checkpoint/recovery costs are increasing when we go to higher levels: $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_k$, $C^{(1)} \leq C^{(2)} \leq \cdots \leq C^{(k)}$, and $R^{(1)} \leq R^{(2)} \leq \cdots \leq R^{(k)}$. The problem is to determine the optimal locations to place checkpoints of various levels in a High-Performance Computing (HPC) application.

In addition to fail-stop errors, *silent* errors, also known as silent data corruptions (such as soft faults in cache or ALU, double bit flips), constitute another threat that cannot be ignored any longer [38, 40, 49, 50]. In contrast to a fail-stop error whose detection is immediate, a silent error is identified only when the corrupted data leads to an unusual application behavior. To deal with silent errors, a traditional checkpointing strategy can still be used, provided that it is coupled with a verification mechanism to detect silent errors [12, 22, 43]. Such a verification mechanism can be either general-purpose (e.g., based on replication [30] or even triplication [37]) or application-specific (e.g., based on algorithm-based fault tolerance (ABFT) [16], on approximate re-execution for ordinary and partial differential equation solvers [13], or on orthogonality checks for Krylov-based sparse solvers [22, 43]). Because verification mechanisms can be costly, alternative techniques capable of rapidly detecting silent errors, with the risk of missing some errors, have been recently developed and studied [3, 5, 14, 20]. We call these verifications *partial verifications*, while perfect verifications (with no error missed) are referred to as *guaranteed verifications*. Furthermore, rather than checkpointing on stable storage (e.g., an external disk), a lightweight mechanism of in-memory checkpoints can be provided to cope with silent errors: one keeps a local copy of the data that has not been corrupted when a silent error strikes, and it can be used to perform a recovery rapidly. However, such local copies are lost once a fail-stop error occurs, and hence checkpoints on stable storage must also be provided when dealing with both sources of errors.

Designing resilience algorithms by combining all of these techniques is quite challenging. In this paper, we deal with a simplified, yet realistic, application framework, where a set of application workflows exchange data at the end of their execution. Such a framework can be modeled as a task graph whose dependencies form a linear chain. This scenario corresponds to an HPC application whose workflow is partitioned into a succession of (typically large) tightly-coupled computational kernels, each of which is identified as a task. Hence, we consider a linear chain of tasks $T_1 \rightarrow T_2 \rightarrow \cdots \rightarrow T_n$, where each task T_i $(1 \leq i \leq n)$ has a weight w_i corresponding to its computational load. The following summarizes our approach to enforcing resilience in this simplified application framework:

Silent errors. To cope with silent errors, we couple in-memory checkpoints with both partial and guaranteed verifications. At the end of each task, we can perform either a partial verification (with cost V) or a guaranteed verification (with cost V^*) of the task output; or, probably less frequently, we can perform a guaranteed verification followed by a memory checkpoint (with cost C^M). Note that we do not take the risk of storing a corrupted checkpoint, hence the need for a guaranteed verification.

Fail-stop errors. To cope with fail-stop errors, we use general multi-level checkpointing and schedule checkpoints of various levels at the end of carefully selected tasks. Checkpoints of level 1 are inserted more frequently than checkpoints of level 2, which themselves are more frequent than checkpoints of level 3, and so on. In our approach, assuming that all checkpointing levels are used, a checkpoint at level ℓ is always preceded by checkpoints at all lower levels 1 to $\ell - 1$, which makes good sense in practice (e.g., with two levels, say local SSD and remote disk, one writes the data onto the local SSD before transferring it to remote the disk). In this context, the checkpointing cost $C^{(\ell)}$ at level ℓ is the cost paid to save data when going from level $\ell - 1$ to level ℓ.

Both error sources. To cope with both fail-stop and silent errors, we combine all these techniques: partial and guaranteed verifications, in-memory checkpointing, and several additional levels of checkpointing.

The main contributions of this paper are several sophisticated dynamic programming algorithms that return the optimal solution for each of the three problems above, i.e., the solution that minimizes the expected execution time of the task chain in polynomial time. To the best of our knowledge, this is the first paper that combines multi-level checkpointing with guaranteed and partial verifications to deal with both fail-stop and silent errors in linear workflows. Furthermore, we present extensive simulations that demonstrate the usefulness of mixing these techniques, and, in particular, we demonstrate the gain obtained thanks to additional verifications and multi-level checkpointing. We show that it may be beneficial to use only some of the checkpointing levels; in this case they are renumbered from 1 to k. The best combination of levels to use can be found by an exhaustive search, since the number of levels k is usually small (3 or 4).

The rest of this paper is organized as follows. We present the dynamic programming algorithm for silent errors with memory checkpoints and verifications in Section 2 and that for fail-stop errors in Section 3. The solution to deal with both error sources is described in Section 4. Simulation results are presented in Section 5. We survey related work in Section 6. Finally, we give concluding remarks and hints for future work in Section 7.

2 Memory checkpointing and verifications for silent errors

In this section, we present a sophisticated dynamic programming algorithm to decide which tasks to checkpoint and which tasks to verify. We first introduce the model in Section 2.1. We describe in Section 2.2 a dynamic programming algorithm for the case where only verified memory checkpoints are taken (i.e., memory checkpoints preceded by a guaranteed verification). We show how to extend this algorithm to add additional guaranteed verifications between checkpoints in Section 2.3, and finally we deal with the more complex case of partial verifications in Section 2.4.

2.1 Model

We consider a chain $T_1 \rightarrow T_2 \rightarrow \cdots \rightarrow T_n$ of n tasks that execute on a large-scale platform subject to silent errors. Each task T_i is associated with a computational load or weight w_i, which is assumed to be known to the algorithm. For notational convenience, we define $W_{]i,j]} = \sum_{p=i+1}^{j} w_p$ to be the total weight of tasks T_{i+1} to T_j for any $0 \leq i < j \leq n$. The arrival times of silent errors follow a *Poisson process* with error rate λ_s. Unlike fail-stop errors, silent errors do not destroy the memory content when they strike. Hence, we can cope with silent errors by using lightweight memory checkpoints. When a silent error is detected, either by a partial verification or by a guaranteed one, we roll back to the nearest memory checkpoint, and recover from the memory copy there. This is much cheaper than checkpointing on and recovering from a disk checkpoint. We enforce that a guaranteed verification is always taken immediately before each memory checkpoint, so that all checkpoints are valid, and hence only one checkpoint needs to be maintained at any time during the execution of the application. Furthermore, we assume that the costs of checkpointing, recovery and verifications are uniform across different tasks, and that they are protected from faults (i.e., silent errors only strike the computations).

Let C^M denote the cost of memory checkpointing and R^M the cost of memory recovery. Also, let V^* denote the cost of guaranteed verification and V the cost of a partial verification. The accuracy of a partial verification is measured by its *recall*, which is denoted by r and represents the proportion of detected errors over all silent errors that have occurred during the execution.[1] For notational convenience, we define $g = 1 - r$ to be the proportion of undetected errors. Note that the guaranteed verification can be considered as one with recall $r^* = 1$. Since a partial verification usually incurs a much smaller cost and yet has a reasonable recall [5, 14], it is highly attractive for detecting silent errors, and we make use of them between guaranteed verifications. For convenience, we introduce before task T_1 a *virtual* task T_0, which is checkpointed, and whose recovery cost is zero. This accounts for the fact that it is always possible to restart the application from scratch (i.e., recover from T_0) at no extra cost.

The Silent problem consists in finding the optimal set of tasks to checkpoint as well as the optimal set of tasks to verify, along with the type of verification (guaranteed or partial) that should be applied. The objective is to minimize the total expected execution time of the task chain.

2.2 With memory checkpoints only

In this section, we present a dynamic programming algorithm when using only verified memory checkpoints (memory checkpoints preceded by a guaranteed verification). A naive brute-force algorithm would have to try all possible solutions (i.e., deciding whether or not to add a checkpoint after each task), resulting in 2^n operations. However, by remarking that the problem can be divided into independent subproblems, we can compute the optimal solution in polynomial time:

Theorem 1. *The optimal solution to the* Silent *problem with only memory checkpoints can be obtained using a dynamic programming algorithm in $O(n^2)$ time and $O(n)$ space, where n is the number of tasks in the chain.*

The remainder of this section is devoted to proving this theorem. We first detail how the dynamic programming algorithm places the memory checkpoints in Section 2.2.1, and then we detail the computation of expected execution time between two memory checkpoints in Section 2.2.2. Finally, we provide the algorithm complexity in Section 2.2.3.

2.2.1 Placing memory checkpoints

We define $\mathbb{E}_{mem}(m_2)$ as the optimal expected time needed to successfully execute all tasks from T_1 to T_{m_2}, where there is a verified memory checkpoint after task T_{m_2}. The goal is to obtain:

$$\mathbb{E}_{mem}(n) ,$$

which is the optimal expected execution time needed to successfully execute all the tasks in the chain. Intuitively, the idea is to compute $\mathbb{E}_{mem}(0), \mathbb{E}_{mem}(1), \mathbb{E}_{mem}(2), \ldots, \mathbb{E}_{mem}(n)$, in this order,

[1] Another measure of accuracy for a partial verification is *precision*, which is denoted as p and represents the proportion of true errors over all silent errors that are reported by the verification. Typically, a tradeoff exists between the recall and precision in the design of a partial verification mechanism. In this paper, we assume perfect precision, i.e., $p = 1$, which has been shown to represent the most useful configuration for optimizing the execution overhead [4].

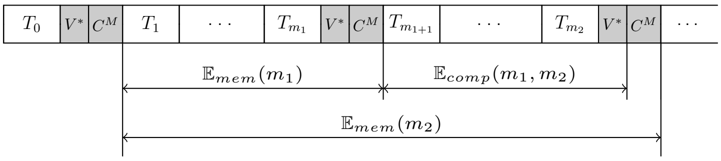


Figure 1: Placing verified memory checkpoints only: m_2 is fixed, and we try all possible locations for an additional verified memory checkpoint at m_1 between T_0 and T_{m_2}. Note that all subproblems $\mathbb{E}_{mem}(m_1)$, with $0 \leq m_1 < m_2$, have already been computed.

so that we can compute each new value by reusing previously computed optimal results. We use memorization: we store and reuse solutions to subproblems instead of recomputing them. This is possible because $\mathbb{E}_{mem}(i)$, for $0 \leq i < n$, does not depend on tasks T_{i+1} to T_n. In other words, $\mathbb{E}_{mem}(i)$ can be used as an independent subproblem, which we compute once and then reuse to solve all $\mathbb{E}_{mem}(j)$ problems, with $i < j \leq n$.

To compute the general subproblem $\mathbb{E}_{mem}(m_2)$, we try all possible locations for an additional intermediate verified memory checkpoint between tasks T_0 and T_{m_2}, as illustrated in Figure 1. For each possible location m_1, we can reuse the optimal result given by $\mathbb{E}_{mem}(m_1)$, and we call $\mathbb{E}_{comp}(m_1, m_2)$, the expected time needed to successfully execute tasks T_{m_1+1} to T_{m_2}, knowing that there is no intermediate checkpoint. Finally, we add the cost of the checkpoint C^M following T_{m_2} (note that we account for the cost of the verification in $\mathbb{E}_{comp}(m_1, m_2)$), and we can write:

$$\mathbb{E}_{mem}(m_2) = \min_{0 \leq m_1 < m_2} \left\{ \mathbb{E}_{mem}(m_1) + \mathbb{E}_{comp}(m_1, m_2) \right\} + C^M \ ,$$

which is initialized with:

$$\mathbb{E}_{mem}(0) = 0 \ .$$

Indeed, when $m_2 = 0$, there is no task to execute, and no room for extra checkpoints.

2.2.2 Computing $\mathbb{E}_{comp}(m_1, m_2)$

Now, to compute the expected time needed to successfully execute several tasks between two verified memory checkpoints, we need only the position of the last verified memory checkpoint m_1, and the position of the next verified memory checkpoint m_2.

First, we pay the cost $W_{]m_1,m_2]}$ to execute all the tasks from T_{m_1+1} to T_{m_2}. Then, we pay the cost of the guaranteed verification V^*. There is a probability $p^s_{]m_1,m_2]} = 1 - e^{-\lambda_s W_{]m_1,m_2]}}$ of detecting a silent error, in which case we recover from the last verified memory checkpoint at m_1, with cost R^M (set to 0 if $m_1 = 0$) and we re-execute all the tasks from there, which is simply $\mathbb{E}_{comp}(m_1, m_2)$. Therefore, we have:

$$\mathbb{E}_{comp}(m_1, m_2) = W_{]m_1,m_2]} + V^* + p^s_{]m_1,m_2]} \left(R^M + \mathbb{E}_{comp}(m_1, m_2) \right) \ .$$

Simplifying the equation above and solving for $\mathbb{E}_{comp}(m_1, m_2)$, we obtain:

$$\mathbb{E}_{comp}(m_1, m_2) = e^{\lambda_s W_{]m_1,m_2]}} \left(W_{]m_1,m_2]} + V^* \right) + \left(e^{\lambda_s W_{]m_1,m_2]}} - 1 \right) R^M \ .$$

2.2.3 Complexity

The complexity is dominated by the computation of the table $\mathbb{E}_{mem}(m_1)$, which contains $O(n)$ entries, and each entry depends on at most n other entries that are already computed. Hence, the overall complexity of the algorithm is $O(n^2)$ in time and $O(n)$ in space. Note that each entry is computed only once using memoization, a well-known technique in dynamic programming [23] that leads to a recursive algorithm whose cost is the same as its iterative counterpart.

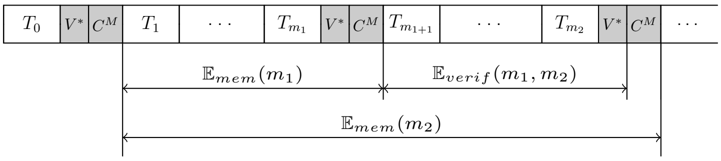


Figure 2: Placing memory checkpoints (with guaranteed verifications): m_2 is fixed, and we try all possible locations for m_1. Note that all subproblems $\mathbb{E}_{mem}(m_1)$, with $0 \leq m_1 < m_2$, have already been computed, while $\mathbb{E}_{verif}(m_1, m_2)$ is computed by yet another dynamic programming level to be described later (see Figure 3).

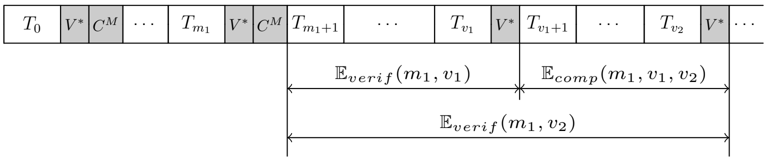


Figure 3: Placing guaranteed verifications: m_1 and v_2 are fixed, and we try all possible locations for an additional guaranteed verification at v_1 between T_{m_1} and T_{v_2}. Note that all subproblems $\mathbb{E}_{verif}(m_1, v_1)$, with $m_1 \leq v_1 < v_2$, have already been computed.

2.3 With memory checkpoints and guaranteed verifications

In this section, we extend the dynamic programming algorithm presented in Section 2.2 to allow for additional intermediate guaranteed verifications between two (verified) memory checkpoints:

Theorem 2. *The optimal solution to the* SILENT *problem with memory checkpoints and intermediate guaranteed verifications can be obtained using a dynamic programming algorithm in $O(n^3)$ time and $O(n^2)$ space, where n is the number of tasks in the chain.*

Figures 2 and 3 illustrate the idea of the algorithm, which now contains two dynamic programming levels, responsible for placing memory checkpoints (Figure 2) and guaranteed verifications (Figure 3), respectively. In the first level, $\mathbb{E}_{comp}(m_1, m_2)$ is replaced by $\mathbb{E}_{verif}(m_1, m_2)$, which accounts for additional guaranteed verifications. The remainder of this section is devoted to proving this theorem.

2.3.1 Placing memory checkpoints

The first level is the same as before (see Section 2.2.1), except that instead of calling $\mathbb{E}_{comp}(m_1, m_2)$ to compute the expected execution time between two memory checkpoints, we now call $\mathbb{E}_{verif}(m_1, m_2)$ to try and place additional guaranteed verifications in this interval (see Figure 2). We can write:

$$\mathbb{E}_{mem}(m_2) = \min_{0 \leq m_1 < m_2} \left\{ \mathbb{E}_{mem}(m_1) + \mathbb{E}_{verif}(m_1, m_2) \right\} + C^M ,$$

which is initialized by:

$$\mathbb{E}_{mem}(0) = 0 .$$

2.3.2 Placing guaranteed verifications

The second level searches where to insert additional guaranteed verifications between the last memory checkpoint at m_1 and the next guaranteed verification at v_2. We define $\mathbb{E}_{verif}(m_1, v_2)$ as the expected execution time needed to sucessfully execute tasks from T_{m_1+1} to T_{v_2}. The function is first called from the first level between two memory checkpoints with $v_2 = m_2$ (as $\mathbb{E}_{verif}(m_1, m_2)$), each of which also comes with a guaranteed verification. The approach to solve the problem is the same as before: suppose $m_1 = 0$, we compute all $\mathbb{E}_{verif}(0, 1), \mathbb{E}_{verif}(0, 2), \mathbb{E}_{verif}(0, 3), \ldots, \mathbb{E}_{verif}(0, n)$, in this order, so that we can compute each step by reusing all previously computed optimal results as before. But because we have n possible values for m_1, we effectively need to solve this problem n times.

Again, this is possible because $\mathbb{E}_{verif}(m_1, v_2)$, $0 \leq m_1 < v_2 < n$ does not depend upon tasks T_{v_2+1} to T_n. In other words, $\mathbb{E}_{verif}(m_1, v_2)$ can be used as an independent subproblem, which is computed exactly once for each (m_1, v_2). These values are stored in a 2D table, so that we can reuse them to solve the larger problems as we progress from T_0 to T_n.

In order to solve $\mathbb{E}_{verif}(m_1, v_2)$, we try all possible locations for the last verification between T_{m_1} and T_{v_2}, and for each possible location v_1, we can reuse the optimal result given by $\mathbb{E}_{verif}(m_1, v_1)$, and we need to compute the expected time needed to successfully execute the tasks between two guaranteed verifications from T_{v_1+1} to T_{v_2}, denoted by $\mathbb{E}_{comp}(m_1, v_1, v_2)$. Therefore, we express $\mathbb{E}_{verif}(m_1, v_2)$ as follows:

$$\mathbb{E}_{verif}(m_1, v_2) = \min_{m_1 \leq v_1 < v_2} \left\{ \mathbb{E}_{verif}(m_1, v_1) + \mathbb{E}_{comp}(m_1, v_1, v_2) \right\} , \tag{1}$$

which is initialized by:

$$\mathbb{E}_{verif}(m_1, m_1) = 0 ,$$

since there is no task to execute and no room for additional guaranteed verifications. Finally, note that we omit the cost of the guaranteed verification after T_{v_2} here, because it is accounted for in the function $\mathbb{E}_{comp}(m_1, v_1, v_2)$.

2.3.3 Computing $\mathbb{E}_{comp}(m_1, v_1, v_2)$

In order to compute the expected time needed to successfully execute tasks between two verifications at v_1 and v_2, we also need to remember the position of the last memory checkpoint at m_1.

First, we pay $W_{]v_1,v_2]}$, which is the time needed to execute tasks from T_{v_1+1} to T_{v_2}, and we account for the cost of the guaranteed verification V^*. Then, there is a probability $p^s_{]v_1,v_2]} = 1 - e^{-\lambda_s W_{]v_1,v_2]}}$ of detecting a silent error, in which case we recover from the last memory checkpoint with a cost R^M (set to 0 if $m_1 = 0$) and only re-execute the tasks from there, using the already computed $\mathbb{E}_{verif}(m_1, v_1)$ followed by $\mathbb{E}_{comp}(m_1, v_1, v_2)$ as before. Therefore, we obtain:

$$\mathbb{E}_{comp}(m_1, v_1, v_2) = W_{]v_1,v_2]} + V^* + p^s_{]v_1,v_2]} \left(R^M + \mathbb{E}_{verif}(m_1, v_1) + \mathbb{E}_{comp}(m_1, v_1, v_2) \right) .$$

Simplifying the equation above and solving for $\mathbb{E}_{comp}(m_1, v_1, v_2)$, we obtain:

$$\begin{aligned} \mathbb{E}_{comp}(m_1, v_1, v_2) = {} & e^{\lambda_s W_{]v_1,v_2]}} \left(W_{]v_1,v_2]} + V^* \right) \\ & + \left(e^{\lambda_s W_{]v_1,v_2]}} - 1 \right) \left(R^M + \mathbb{E}_{verif}(m_1, v_1) \right) . \end{aligned}$$

2.3.4 Complexity

The complexity is now dominated by the computation of the 2D table $\mathbb{E}_{verif}(m_1, v_2)$, which contains $O(n^2)$ entries, and each entry depends on at most n other entries that are already computed. Hence, the overall complexity of the algorithm is $O(n^3)$ in time and $O(n^2)$ in space.

2.4 With partial verifications

It may be beneficial to further add partial verifications between two guaranteed verifications. The intuitive idea would be to add yet another level to the dynamic programming algorithm, and to replace $\mathbb{E}_{comp}(m_1, v_1, v_2)$ in Equation (1) by a call to a function $\mathbb{E}^{(intuitive)}_{partial}(m_1, v_1, p_2, v_2)$, with $p_2 = v_2$, which would compute the optimal expected time needed to execute all the tasks from T_{v_1+1} to T_{p_2} successfully (accounting for errors and re-executions) and add further partial verifications between v_1 and p_2.

However, while the dynamic programming approach was rather intuitive with guaranteed verifications, the problem becomes much harder when partial verifications come into play. Indeed, while computing $\mathbb{E}_{partial}^{(intuitive)}(m_1, v_1, p_2, v_2)$, there is a probability g that an error remains undetected after p_2. When this happens, we need to account for the time lost executing the following tasks until the error is eventually detected by the subsequent partial verifications, or in the worst case by the guaranteed verification at v_2. This is only possible if we know the optimal positions of the partial verifications between p_2 and v_2. This requires the dynamic programming algorithm to first compute the values on the right of p_2, hence progressing the opposite way as what has been done so far.

In other words, instead of calling $\mathbb{E}_{partial}^{(intuitive)}(m_1, v_1, p_2, v_2)$ (with $p_2 = v_2$) on the first call to place additional partial verifications between v_1 and p_2 (on the left of p_2), we now call $\mathbb{E}_{partial}(m_1, v_1, p_1, v_2)$ (with $p_1 = v_1$ on the first call) to place additional partial verifications between p_1 and v_2 (on the right of p_1), as illustrated in Figure 4. $\mathbb{E}_{partial}(m_1, v_1, p_1, v_2)$ is then the optimal expected time needed to execute all the tasks from T_{p_1+1} to T_{v_2}, where T_{p_1} is followed by a partial verification (with the exception of the first call where $p_1 = v_1$) and T_{v_2} is followed by a guaranteed verification, knowing the position of the last memory checkpoint at m_1 and the position of the last guaranteed verification at v_1.

However, this generates another problem: when an error occurs between p_1 and v_2, we need to account for the time lost re-executing all tasks between m_1 and v_1 (which is already computed in $\mathbb{E}_{verif}(m_1, v_1)$) and from v_1 to p_1, which is only possible if we know the optimal positions of the partial verifications between v_1 and p_1 (on the left of p_1). Since we are now solving the sub-problems on the right of p_1 first, we do not know these locations yet.

There is still hope. We make the following simple observation about the re-execution cost: if an error occurs between p_1 and v_2, or occurs earlier and is not detected by p_1, we will re-execute tasks between m_1 and v_1, and between v_1 and p_1, *regardless* of the number and positions of partial verifications between p_1 and v_2. Indeed, the error will be detected either sooner by a partial verification, or later by the guaranteed verification after T_{v_2}. In other words, the expected number of times that tasks between v_1 and p_1 are re-executed due to errors between p_1 and v_2 does not depend on the number and positions of partial verifications between p_1 and v_2: the only thing that matters is that the error will be detected eventually, and the task will be re-executed.

This leads us to the following approach: while deciding the optimal positions of partial verifications between p_1 and v_2, we can ignore the re-execution cost of tasks between v_1 and p_1: these tasks will be re-executed the same number of times in expectation, regardless of the decision we make here, and the total re-execution cost due to errors between p_1 and v_2 is only affected by the positions of p_1 and v_2. Therefore, we account for this cost later, while deciding the optimal position of p_2 between p_1 and v_2, knowing how many times we will execute (and re-execute) tasks between p_1 and p_2 in total (see Lemma 1 for additional details), effectively making $\mathbb{E}_{partial}(m_1, v_1, p_1, v_2)$ an independent subproblem.

Altogether, the following theorem presents a sophisticated dynamic programming algorithm when using partial verifications:

Theorem 3. *The optimal solution to the* Silent *problem while using partial verifications can be obtained using a dynamic programming algorithm in $O(n^5)$ time and $O(n^4)$ space, where n is the number of tasks in the chain.*

The remainder of this section is devoted to proving this theorem. The first two levels of this dynamic programming algorithm, i.e., placing memory checkpoints and guaranteed verifications, are exactly the same as the ones presented in Theorem 2, except that we replace the call to $\mathbb{E}_{comp}(m_1, v_1, v_2)$ by a call to $\mathbb{E}_{partial}(m_1, v_1, v_1, v_2)$, as we did before with guaranteed verifications between memory checkpoints (see Section 2.3.1). The following describes the additional steps required in order to place partial verifications.

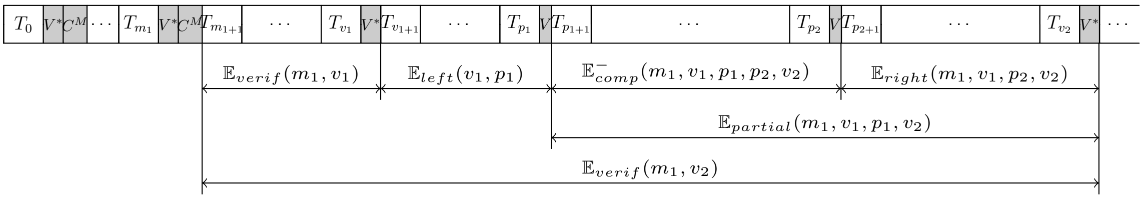

T_0
V^*
C^M
$\cdots$
T_{m_1}
V^*
C^M
$T_{m_{1+1}}$
$\cdots$
T_{v_1}
V^*
$T_{v_{1+1}}$
$\cdots$
T_{p_1}
V
$T_{p_{1+1}}$
$\cdots$
T_{p_2}
V
$T_{p_{2+1}}$
$\cdots$
T_{v_2}
V^*
$\cdots$
$\mathbb{E}_{verif}(m_1, v_1)$
$\mathbb{E}_{left}(v_1, p_1)$
$\mathbb{E}^-_{comp}(m_1, v_1, p_1, p_2, v_2)$
$\mathbb{E}_{right}(m_1, v_1, p_2, v_2)$
$\mathbb{E}_{partial}(m_1, v_1, p_1, v_2)$
$\mathbb{E}_{verif}(m_1, v_2)$

Figure 4: Placing partial verifications: m_1, v_1, p_1 and v_2 are fixed, and we try all possible locations for p_2. Here, both $\mathbb{E}_{verif}(m_1, v_1)$ and $\mathbb{E}_{right}(m_1, v_1, p_2, v_2)$, with $v_1 < p_2 \leq v_2$, have already been computed, which makes it possible to compute $\mathbb{E}_{comp}(m_1, v_1, p_1, p_2, v_2)$, and then $\mathbb{E}_{partial}(m_1, v_1, p_1, v_2)$. Note that we do not need $\mathbb{E}_{left}(v_1, p_1)$ (see Section 2.4 and Lemma 1).

2.4.1 Placing partial verifications

Let $\mathbb{E}_{partial}(m_1, v_1, p_1, v_2)$ denote the optimal expected time needed to execute all the tasks from T_{p_1+1} to T_{v_2}, where T_{p_1} is followed by a partial verification (with the exception of the first call where $p_1 = v_1$) and T_{v_2} is followed by a guaranteed verification, knowing the position of the last memory checkpoint at m_1 and the position of the last guaranteed verification at v_1.

This is yet another level of dynamic programming: suppose that $m_1 = v_1 = 0$ and $v_2 = n$, the goal is to compute $\mathbb{E}_{partial}(0, 0, n, n), \mathbb{E}_{partial}(0, 0, n-1, n), \ldots, \mathbb{E}_{partial}(0, 0, 0, n)$, in this particular order (which is mandatory as shown below in Section 2.4.2) so that at each step, we can reuse all previously computed optimal results. Clearly, we will have to solve this problem $O(n^3)$ times, for all possible $0 \leq m_1 \leq v_1 < v_2 \leq n$.

Similarly as what was done before, we compute $\mathbb{E}_{partial}(m_1, v_1, p_1, v_2)$ by deciding where to place an additional partial verification between tasks T_{p_1+1} and T_{v_2}, and we can write:

$$\mathbb{E}_{partial}(m_1, v_1, p_1, v_2) = \min_{p_1 < p_2 \leq v_2} \left\{ \mathbb{E}_{comp}(m_1, v_1, p_1, p_2, v_2) + \mathbb{E}_{partial}(m_1, v_1, p_2, v_2) \right\} ,$$

where $\mathbb{E}_{comp}(m_1, v_1, p_1, p_2, v_2)$ denotes the expected time needed to successfully execute all the tasks from T_{p_1+1} to T_{p_2}, and the initialization is:

$$\mathbb{E}_{partial}(m_1, v_1, v_2, v_2) = 0 ,$$

because there is no task to execute and no more room for addition partial verifications.

2.4.2 Computing $\mathbb{E}_{comp}(m_1, v_1, p_1, p_2, v_2)$

Recall that $\mathbb{E}_{comp}(m_1, v_1, p_1, p_2, v_2)$ denotes the expected time needed to successfully execute all the tasks from T_{p_1+1} to T_{p_2}, accounting for the time lost due to errors and re-executions, with no undetected silent error after T_{p_2}, knowing that the last memory checkpoint is after T_{m_1}, the last guaranteed verification is after T_{v_1}, and the next guaranteed verification is after T_{v_2}. In order to compute $\mathbb{E}_{comp}(m_1, v_1, p_1, p_2, v_2)$, we need to introduce $\mathbb{E}_{left}(v_1, p_1)$, the expected time needed to successfully execute tasks T_{v_1+1} to T_{p_1}, and $\mathbb{E}_{right}(m_1, v_1, p_2, v_2)$, the expected time lost executing the tasks following T_{p_2}, knowing that *there is an undetected silent error*. Note that in the worst scenario, a silent error will always be detected by the guaranteed verification after T_{v_2}.

Figure 4 shows all the tasks involved in the computation between two partial verifications at p_1 and p_2. Intuitively, the execution goes as follows. First, we execute all the tasks from T_{p_1+1} up to the next partial verification after T_{p_2}, and we pay $W_{]p_1,p_2]}$. Then, we add the cost V for the partial verification, and there is a probability $p^s_{]p_1,p_2]}$ of having a silent error. On the one hand, there is a probability $1 - g$ to detect the error right after the partial verification at T_{p_2}. In this case, we pay a recovery cost R^M from the last memory checkpoint and re-execute the tasks from there by calling $\mathbb{E}_{verif}(m_1, v_1)$, followed by $\mathbb{E}_{left}(v_1, p_1)$ and $\mathbb{E}_{comp}(m_1, v_1, p_1, p_2, v_2)$. However, if the error is not detected (with probability g), we also account for the cost of re-executing tasks from the last memory checkpoint until T_{p_2}, but we further use $\mathbb{E}_{right}(m_1, v_1, p_2, v_2)$ to compute the expected time lost executing the tasks following T_{p_2}, knowing that there is an undetected silent error. In this case, the recovery cost will be accounted for in $\mathbb{E}_{right}$. Therefore, we have:

$$\begin{aligned}\mathbb{E}_{comp}(m_1, v_1, p_1, p_2, v_2) = W_{]p_1,p_2]} + V + p^s_{]p_1,p_2]} \Big(&\mathbb{E}_{verif}(m_1, v_1) \\ &+ \mathbb{E}_{left}(v_1, p_1) + \mathbb{E}_{comp}(m_1, v_1, p_1, p_2, v_2) \\ &+ (1-g)R^M + g\mathbb{E}_{right}(m_1, v_1, p_2, v_2)\Big) \ .\end{aligned}$$

Simplifying the equation above, we obtain:

$$\begin{aligned}\mathbb{E}_{comp}(m_1, v_1, p_1, p_2, v_2) = {} & e^{\lambda_s W_{]p_1,p_2]}} \left(W_{]p_1,p_2]} + V\right) \\ &+ \left(e^{\lambda_s W_{]p_1,p_2]}} - 1\right) \left(\mathbb{E}_{verif}(m_1, v_1) + \mathbb{E}_{left}(v_1, p_1)\right) \\ &+ \left(e^{\lambda_s W_{]p_1,p_2]}} - 1\right) \left((1-g)R^M + g\mathbb{E}_{right}(m_1, v_1, p_2, v_2)\right) \ . \end{aligned} \tag{2}$$

Now, due to the order in which we solve the subproblems, both $\mathbb{E}_{verif}(m_1, v_1)$ and $\mathbb{E}_{partial}(m_1, v_1, p_2, v_2)$ have already been computed and we can get the optimal positions of partial verifications between T_{p_2+1} and T_{v_2}, which are needed to compute $\mathbb{E}_{right}(m_1, v_1, p_2, v_2)$. However, because we solve the subproblems the opposite way as what was done so far, we do not know (yet) the optimal positions of partial verifications between T_{v_1} and T_{p_1}, which are required to compute $\mathbb{E}_{left}(v_1, p_1)$.

Instead, we remove the term

$$\left(e^{\lambda_s W_{p_1,p_2}} - 1\right) \mathbb{E}_{left}(v_1, p_1)$$

from Equation (2), and introduce the modified expression of $\mathbb{E}_{comp}$, denoted by $\mathbb{E}^-_{comp}$, as follows:

$$\begin{aligned}\mathbb{E}^-_{comp}(m_1, v_1, p_1, p_2, v_2) = {} & e^{\lambda_s W_{]p_1,p_2]}} \left(W_{]p_1,p_2]} + V\right) \\ &+ \left(e^{\lambda_s W_{]p_1,p_2]}} - 1\right) \mathbb{E}_{verif}(m_1, v_1) \\ &+ \left(e^{\lambda_s W_{]p_1,p_2]}} - 1\right) \left((1-g)R^M + g\mathbb{E}_{right}(m_1, v_1, p_2, v_2)\right) \ . \end{aligned} \tag{3}$$

Then, to account for the missing $\mathbb{E}_{left}$, we make use of Lemma 1, which shows that, for any number and position of partial verifications between T_{p_2+1} and T_{v_2}, $\mathbb{E}^-_{comp}(m_1, v_1, p_1, p_2, v_2)$ is executed $e^{\lambda_s W_{]p_2,v_2]}}$ times in expectation. As explained before, the intuition behind this result is that the amount of time tasks T_{v_1} to T_{p_1} will be re-executed due to errors in T_{p_1} to T_{v_2} does not depend upon the positions of intermediate partial verifications (e.g., p_2). For every error that occurs between T_{p_1} and T_{v_2}, these tasks *will* be re-executed *regardless* of the position of p_2 (or any other partial verifications between T_{p_1} and T_{v_2}). Hence, instead of accounting for the execution of T_{p_1} to T_{p_2} just once, we now account for all the times we have to execute, and re-execute them due to errors between T_{p_2} and T_{v_2}, and we obtain:

$$\mathbb{E}_{partial}(m_1, v_1, p_1, v_2) = \min_{p_1 < p_2 \leq v_2} \left\{ \mathbb{E}^-_{comp}(m_1, v_1, p_1, p_2, v_2) \cdot e^{\lambda_s W_{]p_2,v_2]}} + \mathbb{E}_{partial}(m_1, v_1, p_2, v_2) \right\} \ , \tag{4}$$

which is initialized by:

$$\begin{aligned}\mathbb{E}_{partial}(m_1, v_1, v_2, v_2) &= 0 \ ; \\ \mathbb{E}^-_{comp}(m_1, v_1, p_1, v_2, v_2) &= e^{\lambda_s W_{]p_1,v_2]}} \left(W_{]p_1,v_2]} + V^*\right) \\ &\quad + \left(e^{\lambda_s W_{]p_1,v_2]}} - 1\right) \mathbb{E}_{verif}(m_1, v_1) \\ &\quad + \left(e^{\lambda_s W_{]p_1,v_2]}} - 1\right) R^M \ ,\end{aligned}$$

because there is no task to execute and no more room for additional partial verifications, and because when computing $\mathbb{E}^-_{comp}(m_1, v_1, p_1, v_2, v_2)$ with $p_2 = v_2$, we need to account for the cost of the guaranteed verification V^* instead of the partial verification. Indeed, Lemma 1 (see below) proves that for any number of partial verifications between p_2 and v_2, $\mathbb{E}^-_{comp}(m_1, v_1, p_1, p_2, v_2)$ is executed $e^{\lambda_s W_{]p_2,v_2]}}$ times in expectation.

2.4.3 Computing $\mathbb{E}_{right}(m_1, v_1, p_1, v_2)$

Finally, in order to get $\mathbb{E}^{-}_{comp}(m_1, v_1, p_1, v_2, v_2)$, we still need compute $\mathbb{E}_{right}(m_1, v_1, p_1, v_2)$, the *optimal* expected time lost executing the tasks T_{p_1+1} to T_{v_2}, assuming that *there is an undetected silent error* in this interval. This computation uses p_2, the *optimal* position of the partial verification immediately following p_1, and it is computed by the dynamic programming algorithm. Indeed, the partial verification after T_{p_2} may or may not detect the error. If the error is detected, we lose $W_{]p_1,p_2]} + V + R^M$ time, while we use $\mathbb{E}_{right}(m_1, v_1, p_2, v_2)$ if the error remains undetected. Altogether, we have:

$$\mathbb{E}_{right}(m_1, v_1, p_1, v_2) = W_{]p_1,p_2]} + V + (1-g)R^M + g\mathbb{E}_{right}(m_1, v_1, p_2, v_2) \ , \tag{5}$$

where p_2 is the optimal position of the next partial verification knowing that there is a partial verification after T_{p_1} and is obtained by backtracking the last step as follows:

$$p_2 = \underset{p_1 < p_2 \leq v_2}{\arg\min} \left\{ \mathbb{E}^{-}_{comp}(m_1, v_1, p_1, p_2, v_2) \cdot e^{\lambda_s W_{]p_2,v_2]}} + \mathbb{E}_{partial}(m_1, v_1, p_2, v_2) \right\} .$$

Note that both $\mathbb{E}^{-}_{comp}(m_1, v_1, p_1, p_2, v_2)$, which only requires $\mathbb{E}_{right}(m_1, v_1, p_2, v_2)$ to be known, and $\mathbb{E}_{partial}(m_1, v_1, p_2, v_2)$ have already been computed by $\mathbb{E}_{partial}(m_1, v_1, p_1, v_2)$. In addition, recall that the time needed to re-execute the tasks after a recovery is not included here.

The initialization is as follows:

$$\begin{aligned}
&\mathbb{E}_{right}(m_1, v_1, v_2, v_2) = R^M, && \text{if } p_1 = p_2 = v_2 \\
&\mathbb{E}_{right}(m_1, v_1, p_1, v_2) = W_{]p_1,p_2]} + V^* + R^M, && \text{if } p_1 < p_2 \text{ and } p_2 = v_2
\end{aligned}$$

Indeed, when $p_2 = v_2$ there is no task to execute after T_{p_2}, and if there was a silent error, it is immediately detected at v_2 (by guaranteed verification), and we just pay R^M. Then, when $p_1 < p_2$ and $p_2 = v_2$, if an error goes undetected after p_1, we will execute tasks from T_{p_2} to T_{v_2}, and we need to take into account the cost of the guaranteed verification V^* at v_2. Again, the error is detected at v_2 and we pay R^M.

2.4.4 Complexity

Clearly, the complexity is now dominated by the computation of the table $\mathbb{E}_{partial}(m_1, v_1, p_1, v_2)$, which contains $O(n^4)$ entries, and each entry depends on at most n other entries that are already computed. Hence, the overall complexity of the algorithm is $O(n^5)$ in time and $O(n^4)$ in space.

Finally, the following presents the formalization and proof of Lemma 1, which has been used in proving Theorem 3.

Lemma 1. *For any number of partial verifications between p_2 and v_2, $\mathbb{E}^{-}_{comp}(m_1, v_1, p_1, p_2, v_2)$ is executed $e^{\lambda_s W_{]p_2,v_2]}}$ times in expectation.*

Proof. Looking at Equation (3), if there is no partial verification after p_2, then we must execute $\mathbb{E}^{-}_{comp}(m_1, v_1, p_1, p_2, v_2)$ at least once when progressing within the computation. Accounting for the term $\mathbb{E}_{left}$ that was suppressed from the final $\mathbb{E}_{comp}(m_1, v_1, p_2, v_2, v_2)$, we must re-execute $\mathbb{E}^{-}_{comp}(m_1, v_1, p_1, p_2, v_2)$ an additional $e^{\lambda_s W_{]p_2,v_2]}} - 1$ times due to errors occurring in $\mathbb{E}_{comp}(m_1, v_1, p_2, v_2, v_2)$. Overall, the expected number of times $\mathbb{E}^{-}_{comp}(m_1, v_1, p_1, p_2, v_2)$ is executed will be

$$1 + (e^{\lambda_s W_{]p_2,v_2]}} - 1) = e^{\lambda_s W_{]p_2,v_2]}} \ .$$

Now, with one intermediate partial verification p_3 between p_2 and v_2, the same reasoning shows that $\mathbb{E}^{-}_{comp}(m_1, v_1, p_2, p_3, v_2)$ must be executed $e^{\lambda_s W_{]p_3,v_2]}}$ times in expectation. Therefore, $\mathbb{E}^{-}_{comp}(m_1, v_1, p_1, p_2, v_2)$ must be executed once, coming from the initial execution, plus an additional $e^{\lambda_s W_{]p_2,p_3]}} - 1$ times due to the re-executions coming from the $\mathbb{E}_{left}$ term suppressed from $\mathbb{E}_{comp}(m_1, v_1, p_2, p_3, v_2)$, which is itself executed $e^{\lambda_s W_{]p_3,v_2]}}$ times. Finally, we must account for the

$e^{\lambda_s W_{]p_3,v_2]}} - 1$ times coming from the last $\mathbb{E}_{comp}(m_1, v_1, p_3, v_2, v_2)$ as well. Overall, the expected number of times $\mathbb{E}^{-}_{comp}(m_1, v_1, p_1, p_2, v_2)$ is executed will be

$$1 + \left(e^{\lambda_s W_{]p_2,p_3]}} - 1\right) \cdot e^{\lambda_s W_{]p_3,v_2]}} + \left(e^{\lambda_s W_{]p_3,v_2]}} - 1\right) = e^{\lambda_s W_{]p_2,v_2]}} \ .$$

It is straightforward to extend this argument to any number of intervals by induction, assuming that it is true for i intermediate partial verifications $p_1, \ldots, p_i$, followed by a guaranteed verification, and adding a partial verification p_{i+1} between p_i and v_2. The same reasoning holds, which concludes the proof. □

3 Multi-level checkpointing for fail-stop errors

In this section, we present a multi-level dynamic programming algorithm to decide which tasks to checkpoint and at which levels when dealing with fail-stop errors. We first introduce the application and checkpointing models in Section 3.1 before presenting the dynamic programming algorithm in Section 3.2.

3.1 Model

As before, we consider a chain $T_1 \to T_2 \to \cdots \to T_n$ of n tasks that execute on a large-scale platform subject to k levels of fail-stop errors. Recall that the weight w_i of task T_i is known to the algorithm, and $W_{]i,j]} = \sum_{p=i+1}^{j} w_p$ denotes the total weight of tasks T_{i+1} to T_j for any $i < j$. Errors of different levels are assumed to be independent, and their arrivals follow *Poisson process* with error rate λ_ℓ for level ℓ, where $1 \leq \ell \leq k$. There are correspondingly k levels of checkpoints available, and each level ℓ is associated with a checkpointing cost $C^{(\ell)}$ and a recovery cost $R^{(\ell)}$. Typically, error rates are decreasing and checkpoint/recovery costs are increasing when we go to higher levels: $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_k$, $C^{(1)} \leq C^{(2)} \leq \cdots \leq C^{(k)}$, and $R^{(1)} \leq R^{(2)} \leq \cdots \leq R^{(k)}$. A level ℓ error destroys all the checkpoints of lower levels (from 1 to $\ell - 1$) and we need to roll back to a checkpoint of level ℓ or higher for recovery. Similarly, a recovery from a level ℓ checkpoint will restore data from all the lower levels. We assume that the costs of checkpointing and recovery are uniform across different tasks, and that they are protected from faults (i.e., errors only strike the computations).

For convenience, we add again before task T_1 a *virtual* task T_0, which is checkpointed at all levels, and whose checkpointing and recovery costs are always zero. This accounts for the fact that it is always possible to restart the application from scratch (i.e., recover from T_0) with no extra cost. Furthermore, we assume that the last task T_n is also always checkpointed at all levels in order to save the final outcome of the computation.

The Multilevel problem consists in finding the optimal set of tasks that should be checkpointed at each level in order to minimize the total expected execution time of the task chain, accounting for failures and re-executions.

3.2 Dynamic programming algorithm

The main difference between fail-stop and silent error is the detection latency: fail-stop errors typically occur in the middle of the computation, causing the application to crash immediately and losing some data. From the algorithmic perspective, it has two implications: (i) when an error occurs, we only need to account for the time lost since the last available checkpoint; and (ii) we must recover from the right checkpoint level, which depends on the type of the error. Our approach for the Multilevel problem remains similar to the algorithm presented in Section 2.2 by using dynamic programming:

Theorem 4. *The optimal solution to the* Multilevel *problem can be obtained using a dynamic programming algorithm in $O(n^{k+1})$ time and $O(n^k)$ space, where n is the number of tasks in the chain, and k is the number of checkpointing levels available.*

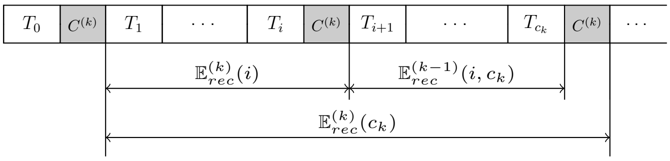

T_0
$C^{(k)}$
T_1
$\cdots$
T_i
$C^{(k)}$
T_{i+1}
$\cdots$
T_{c_k}
$C^{(k)}$
$\cdots$
$\mathbb{E}_{rec}^{(k)}(i)$
$\mathbb{E}_{rec}^{(k-1)}(i, c_k)$
$\mathbb{E}_{rec}^{(k)}(c_k)$

Figure 5: Placing checkpoints at level k: c_k is fixed, and we try all possible locations i for an additional checkpoint at level k between T_0 and T_{c_k}. Note that all subproblems $\mathbb{E}_{rec}^{(k)}(i)$, with $0 \leq i < c_k$, have already been computed, while $\mathbb{E}_{rec}^{(k-1)}(i, c_k)$ is computed by yet another dynamic programming level to be described later (see Figure 6).

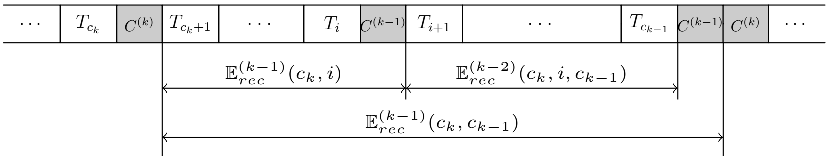


Figure 6: Placing checkpoints at level $k-1$: c_k and c_{k-1} are fixed, and we try possible locations i for an additional checkpoint between T_{c_k} and $T_{c_{k-1}}$. Again, all subproblems $\mathbb{E}_{rec}^{(k-1)}(c_k, i)$, with $c_k \leq i < c_{k-1}$, have already been computed, while $\mathbb{E}_{rec}^{(k-2)}(c_k, i, c_{k-1})$ is computed by the next level of dynamic programming level.

The dynamic programming algorithm consists of k nested levels. The remainder of this section is devoted to proving this theorem by detailing how the k levels of checkpoints are placed.

3.2.1 Placing checkpoints at level k

We start with the highest and most expensive level k. Let $\mathbb{E}_{rec}^{(k)}(c_k)$ denote the optimal expected execution time to successfully execute all tasks from T_1 to T_{c_k} (included), where c_k denotes the index of a task whose output is saved with a level-k checkpoint. Intuitively, we want to obtain:

$$\mathbb{E}_{rec}^{(k)}(n),$$

which is the optimal expected execution time to successfully execute all the tasks in the chain. Backtracking can then be used to get the corresponding optimal set of tasks to checkpoint.

In order to compute $\mathbb{E}_{rec}^{(k)}(c_k)$, we need to decide which tasks to checkpoint at level k between tasks T_0 and T_{c_k} (remember that T_0 is always checkpointed at level k). To this end, we consider each of these tasks as a potential candidate for the *last checkpoint* at level k before T_{c_k}, and return the minimum expected execution time as follows (see Figure 5):

$$\mathbb{E}_{rec}^{(k)}(c_k) = \min_{0 \leq i < c_k} \left\{ \mathbb{E}_{rec}^{(k)}(i) + \mathbb{E}_{rec}^{(k-1)}(i, c_k) \right\} + C^{(k)} \ .$$

For each task T_i, we first call $\mathbb{E}_{rec}^{(k)}(i)$ recursively to decide which additional tasks should be checkpointed at level k, between task T_1 and the newly checkpointed task T_i. Then, we compute the expected execution time between tasks T_i and T_{c_k} by calling the next level function $\mathbb{E}_{rec}^{(k-1)}(i, c_k)$ that decides which tasks to checkpoint at level $k-1$, knowing that both tasks T_i and T_{c_k} are already checkpointed at level k. Finally, we account for the level-k checkpointing cost $C^{(k)}$ after task T_{c_k}.

3.2.2 Placing checkpoints at level $k-1$.

Now, let $\mathbb{E}_{rec}^{(k-1)}(c_k, c_{k-1})$ denote the optimal expected execution time needed to successfully execute all the tasks from T_{c_k} to $T_{c_{k-1}}$ (included), where c_k denotes the position of the last checkpoint at level k, and c_{k-1} denotes the position of the next level $k-1$ checkpoint. Note that the first time we call this function (while computing $\mathbb{E}_{rec}^{(k)}(c_k)$ above), c_{k-1} (a.k.a. c_k above) actually denotes the position the next level-k checkpoint, which by construction always includes a level $k-1$ checkpoint as well, and that is accounted for in the equation below. Similarly to the level-k function, we try all tasks T_i between T_{c_k} and $T_{c_{k-1}}$ (included) for the last checkpoint at level $k-1$, so that we can write (see Figure 6):

$$\mathbb{E}_{rec}^{(k-1)}(c_k, c_{k-1}) = \min_{c_k \leq i < c_{k-1}} \left\{ \mathbb{E}_{rec}^{(k-1)}(c_k, i) + \mathbb{E}_{rec}^{(k-2)}(c_k, i, c_{k-1}) \right\} + C^{(k-1)} \ .$$

We first call $\mathbb{E}_{rec}^{(k-1)}(c_k, i)$ recursively between tasks T_{c_k} and T_i to place additional checkpoints at level $k-1$, then call the function $\mathbb{E}_{rec}^{(k-2)}(c_k, i, c_{k-1})$ to place level $k-2$ checkpoints between tasks T_i and $T_{c_{k-1}}$, and finally account for the level $k-1$ checkpointing cost $C^{(k-1)}$ after task $T_{c_{k-1}}$. Note that T_{c_k} will always be checkpointed at level $k-1$, in addition of the level-k checkpoint that was already placed before. In fact, by following this approach, we guarantee that a high-level checkpoint always includes all the lower-level checkpoints as well.

3.2.3 Placing checkpoints at level ℓ

The function for placing checkpoints at level $k-2$ would now contain three parameters, because we need to remember both c_k and c_{k-1}, the positions of the last checkpoint at level k and level $k-1$, respectively, as well as c_{k-2}, the position of the next checkpoint at the current level $k-2$. This is because in case an error from level k or level $k-1$ strikes, we need to know which is the nearest available checkpoint to recover from.

In general, let $\mathbb{E}_{rec}^{(\ell)}(c_k, \ldots, c_{\ell+1}, c_\ell)$ denote the optimal expected execution time needed to execute tasks $T_{c_{\ell+1}+1}$ to T_{c_ℓ} (included), where $c_{\ell+1}$ is the position of the last checkpoint at level $\ell+1$ and c_ℓ is the position of the next level-ℓ checkpoint. Similarly to the functions at level k and level $k-1$, the goal of this function is to place additional level-ℓ checkpoints between tasks $T_{c_{\ell+1}}$ and T_{c_ℓ}. We denote by i the position of the newly added checkpoint at current level ℓ, and we try all possible positions between $c_{\ell+1}$ and c_ℓ. Hence, we derive:

$$\mathbb{E}_{rec}^{(\ell)}(c_k, \ldots, c_{\ell+1}, c_\ell) = \min_{c_{\ell+1} \leq i < c_\ell} \left\{ \mathbb{E}_{rec}^{(\ell)}(c_k, \ldots, c_{\ell+1}, i) + \mathbb{E}_{rec}^{(\ell-1)}(c_k, \ldots, c_{\ell+1}, i, c_\ell) \right\} + C^{(\ell)} \ . \quad (6)$$

For each candidate T_i, we first call $\mathbb{E}_{rec}^{(\ell)}(c_k, \ldots, c_{\ell+1}, i)$ to place additional level-ℓ checkpoints between tasks $T_{c_{\ell+1}}$ and T_i. Then, we call $\mathbb{E}_{rec}^{(\ell-1)}(c_k, \ldots, c_{\ell+1}, i, c_\ell)$ to place additional level $\ell-1$ checkpoints between tasks T_i and T_{c_ℓ}. Finally, we account for the level-ℓ checkpointing cost $C^{(\ell)}$ after task T_{c_ℓ}.

3.2.4 Initialization

To initialize the dynamic program at each level ℓ, we set:

$$\mathbb{E}_{rec}^{(\ell)}(c_k, \ldots, c_{\ell+1}, c_{\ell+1}) = 0 \ ,$$

which occurs once when $i = c_{\ell+1}$ in Equation (6); there is no task to execute, and the cost of the checkpoint after T_i has been accounted for already. Then, when the last level is reached, i.e., when $\ell = 1$, there is no more checkpointing level to try, and we set:

$$\mathbb{E}_{rec}^{(0)}(c_k, \ldots, c_2, c_1, c_{1'}) = \mathbb{E}_{comp}(c_k, \ldots, c_2, c_1, c_{1'}) \ ,$$

where $\mathbb{E}_{comp}(c_k, \ldots, c_2, c_1, c_{1'})$ denotes the expected execution time needed to execute tasks T_{c_1+1} to $T_{c_{1'}}$ (included), with no additional intermediate checkpoints in between.

3.2.5 Computing $\mathbb{E}_{comp}(c_k, \dots, c_1, c_{1'})$

Given the positions of the checkpoints, we can now compute the actual expected execution time needed to successfully execute the tasks between any two consecutive level-1 checkpoints. We make use of the following well-known properties of independent Poisson processes [31, Chapter 2.3].

Property 1. *During the execution of a sequence of tasks with total work W, let X_ℓ denote the time when the first level-ℓ error strikes. Thus, X_ℓ is a random variable following* exponential *distribution with parameter λ_ℓ, for all $\ell = 1, 2, \dots, k$.*

(1) Let X denote the time when the first error (of any level) strikes. We have $X = \min\{X_1, X_2, \dots, X_k\}$, which follows exponential distribution with parameter $\Lambda = \sum_{\ell=1}^{k} \lambda_\ell$. The probability of having an error (from any level) during the execution is therefore $P(X \le W) = 1 - e^{-\Lambda W}$.

(2) Given that an error (from any level) strikes during the execution of the tasks, the probability that the error belongs to a particular level is proportional to the error rate of that level, i.e., $P(X = X_\ell | X \le W) = \frac{\lambda_\ell}{\Lambda}$, for all $\ell = 1, 2, \dots, k$.

Recall that $W_{]c_1,c_{1'}]} = \sum_{i=c_1+1}^{c_{1'}} w_i$ denotes the total computational load between tasks T_{c_1+1} and $T_{c_{1'}}$. Hence, with probability $p^f_{]c_1,c_{1'}]} = 1 - e^{-\Lambda \cdot W_{]c_1,c_{1'}]}}$, at least one fail-stop error (from any level) *will* occur during the execution of tasks T_{c_1+1} to $T_{c_{1'}}$ (included). When this happens, we first need to account for the time lost during the execution (up to the error), denoted by $T^{\text{lost}}_{]c_1,c_{1'}]}$. Then, we need to roll back to the nearest checkpoint, depending on the level of the error. For example, with probability $\frac{\lambda_3}{\Lambda}$, we need to recover from the last level-3 checkpoint, and we pay $R^{(3)}$, the cost to recover from task T_{c_3} using level-3 recovery. When the recovery is done, we need to re-execute all the tasks, first from T_{c_3+1} to T_{c_2}, then from T_{c_2+1} to T_{c_1}, and finally from T_{c_1} to $T_{c_{1'}}$ again. By construction, there is no other level-3 checkpoint between T_{c_3} and T_{c_2}, so the expected time to re-execute all the tasks up to the next level-2 checkpoint, that is from tasks T_{c_3} to T_{c_2}, is simply $\mathbb{E}^{(2)}_{rec}(c_k, \dots, c_3, c_2)$. Then, we proceed by re-executing the tasks up to the next level-1 checkpoint, which takes $\mathbb{E}^{(1)}_{rec}(c_k, \dots, c_2, c_1)$ time, at which point we can just call $\mathbb{E}_{comp}(c_1, \dots, c_1, c_{1'})$ again, to restart this whole process until the execution of tasks T_{c_1} to $T_{c_{1'}}$ is eventually successful. When no error occurs, which happens with probability $1 - p^f_{]c_1,c_{1'}]}$, we just need to pay the cost of executing all the tasks without error, i.e., $W_{]c_1,c_{1'}]}$. Therefore, we derive:

$$
\begin{aligned}
\mathbb{E}_{comp}(c_k, \dots, c_1, c_{1'}) = \left(1 - p^f_{]c_1,c_{1'}]}\right) W_{]c_1,c_{1'}]} + p^f_{]c_1,c_{1'}]} \Bigg(& T^{\text{lost}}_{]c_1,c_{1'}]} \\
& + \frac{\lambda_1}{\Lambda}\left(R^{(1)} + \mathbb{E}_{comp}(c_k, \dots, c_1, c_{1'})\right) \\
& + \frac{\lambda_2}{\Lambda}\left(R^{(2)} + \mathbb{E}^{(1)}_{rec}(c_k, \dots, c_2, c_1) + \mathbb{E}_{comp}(c_k, \dots, c_1, c_{1'})\right) \\
& \vdots \\
& + \frac{\lambda_{k-1}}{\Lambda}\left(R^{(k-1)} + \sum_{\ell=2}^{k-1} \mathbb{E}^{(\ell-1)}_{rec}(c_k, \dots, c_\ell, c_{\ell-1}) + \mathbb{E}_{comp}(c_k, \dots, c_1, c_{1'})\right) \\
& + \frac{\lambda_k}{\Lambda}\left(R^{(k)} + \sum_{\ell=2}^{k} \mathbb{E}^{(\ell-1)}_{rec}(c_k, \dots, c_\ell, c_{\ell-1}) + \mathbb{E}_{comp}(c_k, \dots, c_1, c_{1'})\right)\Bigg) .
\end{aligned}
$$

Simplifying the equation above, we can obtain:

$$\mathbb{E}_{comp}(c_k, \dots, c_1, c_{1'}) = e^{-\Lambda \cdot W_{]c_1,c_{1'}]}} W_{]c_1,c_{1'}]} + \left(1 - e^{-\Lambda \cdot W_{]c_1,c_{1'}]}}\right) \Bigg(T^{\text{lost}}_{]c_1,c_{1'}]} + \sum_{h=1}^{k} \frac{\lambda_h}{\Lambda} \Big(R^{(h)} + \sum_{\ell=2}^{h} \mathbb{E}^{(\ell-1)}_{rec}(c_k, \dots, c_\ell, c_{\ell-1}) \Big) + \mathbb{E}_{comp}(c_k, \dots, c_1, c_{1'}) \Bigg) . \quad (7)$$

In order to compute the expected execution time, we need to compute $T^{\text{lost}}_{]c_1,c_{1'}]}$, which is the expected time loss due to a fail-stop error occurring during the execution of tasks T_{c_1} to $T_{c_{1'}}$. We obtain:

$$\begin{aligned} T^{\text{lost}}_{]c_1,c_{1'}]} &= \int_0^\infty x \mathbb{P}(X = x | X < W_{]c_1,c_{1'}]}) dx \\ &= \frac{1}{\mathbb{P}(X < W_{]c_1,c_{1'}]})} \int_0^{W_{]c_1,c_{1'}]}} x \mathbb{P}(X = x) dx \ , \end{aligned}$$

where $\mathbb{P}(X = x)$ denotes the probability that a fail-stop error strikes at time x. By definition, we have $\mathbb{P}(X = x) = \Lambda e^{-\Lambda x}$ and $\mathbb{P}(X < W_{]c_1,c_{1'}]}) = 1 - e^{-\Lambda W_{]c_1,c_{1'}]}}$. Integrating by parts, we have:

$$T^{\text{lost}}_{]c_1,c_{1'}]} = \frac{1}{\Lambda} - \frac{W_{]c_1,c_{1'}]}}{e^{\Lambda W_{]c_1,c_{1'}]}} - 1} \ . \quad (8)$$

Now, substituting $T^{\text{lost}}_{]c_1,c_{1'}]}$ above into Equation (7), and solving for $\mathbb{E}_{comp}(c_k, \dots, c_1, c_{1'})$, we obtain:

$$\mathbb{E}_{comp}(c_k, \dots, c_1, c_{1'}) = \frac{e^{\Lambda \cdot W_{]c_1,c_{1'}]}} - 1}{\Lambda} \left(1 + \sum_{h=1}^{k} \lambda_h \Big(R^{(h)} + \sum_{\ell=2}^{h} \mathbb{E}^{(\ell-1)}_{rec}(c_k, \dots, c_\ell, c_{\ell-1}) \Big) \right) .$$

3.2.6 Complexity

The complexity is dominated by the computation of the table $\mathbb{E}^{(1)}_{rec}(c_k, \dots, c_2, c_1)$, which contains n^k entries. In order to compute each entry, a minimum over at most n other entries (that are already computed) is required. All tables are computed in a bottom-up fashion, from the left to the right of the intervals. Hence, the overall complexity of the algorithm is $O(n^{k+1})$.

4 Dealing with both fail-stop and silent errors

On real-life platforms, fail-stop errors and silent errors coexist, and thus resilience algorithms must be able to cope with both error sources simultaneously. In this section, we describe a multi-level dynamic programming algorithm to address this challenging problem.

The new algorithm is a combination of the dynamic programming algorithms presented in the preceding sections. In particular, we place k levels of *disk*[2] checkpoints to deal with different fail-stop errors, followed by another level of memory checkpoints, and additional verifications (guaranteed or partial), to deal with silent errors. We call this problem the MULTILEVEL-SILENT problem, and the objective is to find the optimal positions in the task chain to place different checkpoints (disk and memory) as well as verifications (guaranteed and partial) to minimize the expected execution time. The following theorem presents the solution to this problem.

[2] By *disk*, we mean stable storage devices or advanced checkpointing mechanisms (e.g., partner-copy [7]) that can survive various sources of fail-stop errors, in opposition to *memory*, which can only survive silent errors.

Theorem 5. *The optimal solution to the* Multilevel-Silent *problem can be obtained using a dynamic programming algorithm in $O(n^{k+5})$ time and $O(n^{k+4})$ space, where n is the number of tasks in the chain and k is number of checkpointing levels to handle fail-stop errors.*

Proof. The dynamic programming for fail-stop errors is exactly the same as the one shown in Section 3.2, up to the call to the function $\mathbb{E}^{(0)}_{rec}(c_k, \dots, c_2, c_1, c_{1'})$, which is invoked after placing the last level-1 checkpoints. Now, in order to handle silent errors, we set:

$$\mathbb{E}^{(0)}_{rec}(c_k, \dots, c_2, c_1, c_{1'}) = \mathbb{E}_{mem}(c_k, \dots, c_1, c_{1'}) \ ,$$

where

$$\mathbb{E}_{mem}(c_k, \dots, c_1, m_2) = \min_{c_1 \le m_1 < m_2} \left\{ \mathbb{E}_{mem}(c_k, \dots, c_1, m_1) + \mathbb{E}_{verif}(c_k, \dots, c_1, m_1, m_2) \right\} + C^M \ ,$$

with $m_2 = c_{1'}$ when first called from $\mathbb{E}_{mem}(c_k, \dots, c_1, m_2)$, and

$$\mathbb{E}_{verif}(c_k, \dots, c_1, m_1, v_2) = \min_{m_1 \le v_1 < v_2} \left\{ \mathbb{E}_{verif}(c_k, \dots, c_1, m_1, v_1) + \mathbb{E}_{partial}(c_k, \dots, c_1, m_1, v_1, v_2, v_2) \right\} \ ,$$

with $v_2 = m_2$ when first called from $\mathbb{E}_{verif}(c_k, \dots, c_1, m_1, v_2)$. Overall, $\mathbb{E}_{mem}(c_k, \dots, c_1, m_2)$ and $\mathbb{E}_{verif}(c_k, \dots, c_1, m_1, v_2)$ remain the same as in Section 2.4, except for the fact that we now need to remember the position of the last checkpoint at each level, in case a fail-stop error occurs during the execution of tasks T_{v_1+1} to T_{v_2}. As for the initialization, we set $\mathbb{E}_{mem}(c_k, \dots, c_1, c_1) = 0$ and $\mathbb{E}_{verif}(c_k, \dots, c_1, m_1, m_1) = 0$, which occur once when $m_1 = c_1$ and $v_1 = m_1$, respectively. In both cases, there is no task to execute, and the cost of the checkpoint/verification has already been accounted for.

Placing partial verifications. Similarly, the function to place additional partial verifications becomes $\mathbb{E}_{partial}(c_k, \dots, c_1, m_1, v_1, p_1, v_2)$, and the expected number of times the function $\mathbb{E}^{-}_{comp}(c_k, \dots, c_1, m_1, v_1, p_1, p_2, v_2)$ is executed must now account for both silent errors and fail-stop errors. Hence, we can write:

$$\begin{aligned}\mathbb{E}_{partial}(c_k, \dots, c_1, m_1, v_1, p_1, v_2) = \min_{p_1 < p_2 \le v_2} \Big\{ & \mathbb{E}^{-}_{comp}(c_k, \dots, c_1, m_1, v_1, p_1, p_2, v_2) \cdot e^{(\lambda_s + \Lambda) W_{]p_2, v_2]}} \\ & + \mathbb{E}_{partial}(c_k, \dots, c_1, m_1, v_1, p_2, v_2) \Big\} \ .\end{aligned}$$

Computing $\mathbb{E}^{-}_{comp}(c_k, \dots, c_1, m_1, v_1, p_1, p_2, v_2)$. On the one hand, if a fail-stop error occurs with probability $p^f_{]p_1,p_2]} = 1 - e^{-\Lambda \cdot W_{]p_1,p_2]}}$, we can apply the same method as in Section 3. We recover from the nearest checkpoint depending on the error level, and we re-execute all the tasks up to $\mathbb{E}^{(1)}_{rec}(c_k, \dots, c_1)$, then we call $\mathbb{E}_{mem}(c_k, \dots, c_1, m_1)$, followed by a call to the function $\mathbb{E}_{verif}(c_k, \dots, c_1, m_1, v_1)$ to account for the time needed to re-execute the tasks between the last memory checkpoint after T_{m_1} to the next guaranteed verification after T_{v_1}, and finally we are left with the remaining tasks between T_{v_1+1} and T_{p_1}, and we call $\mathbb{E}^{-}_{comp}(c_k, \dots, c_1, m_1, v_1, p_1, p_2, v_2)$ again. On the other hand, with probability $(1 - p^f_{p_1,p_2})$, there is no fail-stop error. In that case, we execute all the tasks from T_{p_1+1} to the next verification after T_{p_2}, as was done in Section 2.4.

Overall, we can write:

$$\begin{aligned}
&\mathbb{E}^{-}_{comp}(c_k,\dots,c_1,m_1,v_1,p_1,p_2,v_2) = \\
&\qquad p^{f}_{]p_1,p_2]}\Bigg(T^{\text{lost}}_{]p_1,p_2]} + \sum_{h=1}^{k}\frac{\lambda_h}{\Lambda}\Big(R^{(h)} + \sum_{\ell=2}^{h}\mathbb{E}^{(\ell-1)}_{rec}(c_k,\dots,c_\ell,c_{\ell-1})\Big) \\
&\qquad\qquad + \mathbb{E}_{mem}(c_k,\dots,c_1,m_1) + \mathbb{E}_{verif}(c_k,\dots,c_1,m_1,v_1) \\
&\qquad\qquad + \mathbb{E}^{-}_{comp}(c_k,\dots,c_1,m_1,v_1,p_1,p_2,v_2)\Bigg) \\
&\quad + \Big(1-p^{f}_{]p_1,p_2]}\Big)\Bigg(W_{]p_1,p_2]} + V + p^{s}_{]p_1,p_2]}\Big(\mathbb{E}_{verif}(c_k,\dots,c_1,m_1,v_1) \\
&\qquad\qquad + \mathbb{E}^{-}_{comp}(c_k,\dots,c_1,m_1,v_1,p_1,p_2,v_2) \\
&\qquad\qquad + (1-g)R^{M} + g\mathbb{E}_{right}(c_k,\dots,c_1,m_1,v_1,p_2,v_2)\Big)\Bigg)\ .
\end{aligned}$$

Simplifying the equation above and solving for $\mathbb{E}^{-}_{comp}$, we obtain:

$$\begin{aligned}
&\mathbb{E}^{-}_{comp}(c_k,\dots,c_1,m_1,v_1,p_1,p_2,v_2) = \\
&\qquad + e^{\lambda_s W_{]p_1,p_2]}}\left(\frac{e^{\Lambda W_{]p_1,p_2]}}-1}{\Lambda} + V\right) \\
&\qquad + e^{\lambda_s W_{]p_1,p_2]}}\left(e^{\Lambda W_{]p_1,p_2]}}-1\right)\Bigg(\sum_{h=1}^{k}\frac{\lambda_h}{\Lambda}\Big(R^{(h)} + \sum_{\ell=2}^{h}\mathbb{E}^{(\ell-1)}_{rec}(c_k,\dots,c_\ell,c_{\ell-1})\Big) \\
&\qquad\qquad + \mathbb{E}_{mem}(c_k,\dots,c_1,m_1)\Bigg) \\
&\qquad + \Big(e^{(\lambda_s+\Lambda)W_{]p_1,p_2]}}-1\Big)\,\mathbb{E}_{verif}(c_k,\dots,c_1,m_1,v_1) \\
&\qquad + \left(e^{\lambda_s W_{]p_1,p_2]}}-1\right)\Big((1-g)R^{M} + g\mathbb{E}_{right}(c_k,\dots,c_1,m_1,v_1,p_2,v_2)\Big)\ .
\end{aligned}$$

Computing $\mathbb{E}_{right}(c_k,\dots,c_1,m_1,v_1,p_1,v_2)$. Remember that $\mathbb{E}_{right}(c_k,\dots,c_1,m_1,v_1,p_1,v_2)$ denotes the expected time lost executing the tasks T_{p_1+1} to T_{v_2}, assuming that *there was a silent error* in this interval. Equation (5) already accounts for the time lost in that case, but only when there is no fail-stop error. Similarly to $\mathbb{E}^{-}_{comp}$ above, we consider fail-stop errors between T_{p_1+1} and T_{p_2}, because fail-stop errors between T_{p_2+1} and T_{v_2} will be accounted for in $\mathbb{E}_{right}(c_k,\dots,c_1,m_1,v_1,p_2,v_2)$. Note that even if we know that there is a silent error in the interval, we may need to recover from a fail-stop error if it strikes before the silent error is detected. Altogether, we have:

$$\begin{aligned}
&\mathbb{E}_{right}(c_k,\dots,c_1,m_1,v_1,p_1,v_2) = \\
&\quad p^{f}_{]p_1,p_2]}\Bigg(T^{\text{lost}}_{]p_1,p_2]} + \sum_{h=1}^{k}\frac{\lambda_h}{\Lambda}\Big(R^{(h)} + \sum_{\ell=2}^{h}\mathbb{E}^{(\ell-1)}_{rec}(c_k,\dots,c_\ell,c_{\ell-1})\Big) + \mathbb{E}_{mem}(c_k,\dots,c_1,m_1)\Bigg) \\
&\quad + \Big(1-p^{f}_{]p_1,p_2]}\Big)\Big(W_{]p_1,p_2]} + V + (1-g)R^{M} + g\mathbb{E}_{right}(c_k,\dots,c_1,m_1,v_1,p_2,v_2)\Big)\ .
\end{aligned}$$

Finally, simplifying the equation above, we obtain:

$$
\begin{aligned}
&\mathbb{E}_{right}(c_k, \dots, c_1, m_1, v_1, p_1, v_2) = \\
&\quad \left(1 - e^{-\Lambda W_{]p_1,p_2]}}\right) \left(\frac{1}{\Lambda} + \sum_{h=1}^{k} \frac{\lambda_h}{\Lambda} \left(R^{(h)} + \sum_{\ell=2}^{h} \mathbb{E}_{rec}^{(\ell-1)}(c_k, \dots, c_\ell, c_{\ell-1}) \right) + \mathbb{E}_{mem}(c_k, \dots, c_1, m_1) \right) \\
&\quad + e^{-\Lambda W_{]p_1,p_2]}} \left(V + (1-g)R^M + g\mathbb{E}_{right}(c_k, \dots, c_1, m_1, v_1, p_2, v_2) \right) .
\end{aligned}
$$

The initialization remains $\mathbb{E}_{right}(c_k, \dots, c_1, m_1, v_1, v_2, v_2) = R^M$.

Complexity. The complexity is dominated by the computation of the dynamic programming table $\mathbb{E}_{partial}(c_k, \dots, c_1, m_1, v_1, p_1, v_2)$, which contains $O(n^{k+4})$ entries, and each entry depends on at most n other entries that are already computed. Therefore, the complexity of the dynamic programming algorithm to handle both fail-stop and silent errors is $O(n^{k+5})$. □

We point out that, in practical systems, the number of checkpointing levels k is generally quite small and rarely exceeds 3 or 4 [38, 7], while linear application workflows rarely exceed a few tens of tasks. Hence, our algorithm can be efficiently applied to these practical scenarios in reasonable time and space.

5 Performance evaluation

In this section, we conduct a set of simulations to assess the relative efficiency of our approach under practical scenarios. We instantiate the performance model with two different sets of realistic parameters obtained from the literature. The simulation code is publicly available at `http://graal.ens-lyon.fr/~yrobert/chainmultilevel.zip` for interested readers to experiment with their own parameters.

Simulation setup. We make several assumptions on the input parameters. First, the checkpoint and recovery costs both depend on size of the the task output file, and the final cost is mostly determined by the available bandwidth at each level. As such, we make the assumption that the recovery cost for a given level is equivalent to the corresponding checkpointing cost, i.e., $R^{(i)} = C^{(i)}$ for $1 \leq i \leq k$. This is a common assumption [38, 42, 25], even though in practice the recovery cost can be expected to be smaller than the checkpoint cost [25, 26].

Then, we assume that, similarly, a guaranteed verification must check all the data in memory, making its cost in the same order as that of a memory checkpoint, i.e., $V^* = C^M$. Furthermore, we assume partial verifications similar to those proposed in [5, 6, 14], with very low costs while offering good recalls. In the following, we set $V = V^*/100$ and $r = 0.8$. The total computational weight is set to be $W = 25000$ seconds (or $W = 3600s$ in some simulations), and it is distributed among up to $n = 50$ tasks in three different patterns shown as follows.

(1) *Uniform*: all tasks share the same weight W/n, as in matrix multiplication or in some iterative stencil kernels.

(2) *Decrease*: task T_i has weight $\alpha(n+1-i)^2$, where $\alpha \approx 3W/n^3$; this quadratically decreasing function resembles some dense matrix solvers, e.g., by using LU or QR factorization.

(3) *HighLow*: a set of tasks with large weight is followed by a set of tasks with small weight. In the simulations, we set 10% of the tasks to be large and let them contain 60% of the total computational weight.

We point out that all these choices are somewhat arbitrary and can easily be modified in the evaluations; however we believe they represent reasonable values for current and next-generation HPC applications. We first investigate the impact of using guaranteed and partial verifications in Section 5.1, by focusing on a platform with a single level of checkpoints for fail-stop errors. Then, we study the impact of multi-level checkpointing in Section 5.2.

5.1 Results for two-level checkpointing

In this section, we perform a set of experiments based on the characteristics of four platforms taken from the literature. We start by analyzing the combined algorithm, but in a somewhat simplified context, with only one level of checkpoint to deal with fail-stop errors (i.e., $k = 1$). We compare three algorithms: (i) a single-level algorithm A_{DV^*} with only disk checkpoints to handle both fail-stop and silent errors (with additional guaranteed verifications); (ii) a two-level algorithm A_{DMV^*} with additional memory checkpoints for silent errors; and (iii) the combined algorithm A_{DMV} using additional partial verifications. The optimal positions of verifications can be easily derived for A_{DV^*} using a simplification of the proposed dynamic programming algorithm in Section 4, with $k = 1$ level of fail-stop errors and no additional memory checkpoints.

Platform settings. Table 1 presents the four platforms used in the simulations and their main parameters. These platforms have been used to evaluate the Scalable Checkpoint/Restart (SCR) library by Moody et al. [38], who provide accurate measurements for λ_f, λ_s, $C^{(1)}$ and C^M using real applications. Note that in this configuration, $C^{(1)}$ denotes the cost of checkpointing to disk, and is referred to as a disk checkpoint below, as opposed to the memory checkpoint, which is done in RAM. There is an exception with the Coastal platform, on which SSD technology is used for memory checkpointing; this provides more data space, at the cost of higher checkpointing costs. In addition, note that the Hera platform has the worst error rates, with a platform MTBF of 12.2 days for fail-stop errors and 3.4 days for silent errors. In comparison, and despite its higher number of nodes, the Coastal platform features a platform MTBF of 28.8 days for fail-stop errors and 5.8 days for silent errors.

Set	From	Platform	#Nodes	λ_f	λ_s	$C^{(1)}$	C^M
(A)	Moody et al. [38]	Hera	256	9.46e-7	3.38e-6	300s	15.4s
		Atlas	512	5.19e-7	7.78e-6	439s	9.1s
		Coastal	1024	4.02e-7	2.01e-6	1051s	4.5s
		Coastal SSD	1024	4.02e-7	2.01e-6	2500s	180.0s

Table 1: Set of parameters (A) used as input for simulations.

Impact of the number of tasks. The first column of Figure 7 presents, for each platform, the normalized makespan with respect to the error-free execution time for different numbers of tasks with the *Uniform* pattern. Note that varying the number of tasks has an impact on both the size of the tasks and the maximum number of checkpoints and verifications that the scheduling algorithm can place. When the number of tasks is small (e.g., less than 5), the probability of having an error during the execution (either a fail-stop or a silent) increases quickly (more than 10% on Hera) for a single task. As a result, the application experiences more recoveries and re-executions with larger tasks, which increases the execution overhead. However, when the number of tasks is large enough, the size of the tasks becomes small and the probability of having an error during the execution of one task drops significantly, reducing the recovery and re-execution costs at the same time.

Single-level algorithm $\mathbf{A}_{DV^*}$. The second column of Figure 7 shows the numbers of disk checkpoints (with associated memory checkpoints) and guaranteed verifications used by the A_{DV^*} algorithm on the four platforms and for different numbers of tasks. We observe that a large number of guaranteed verifications is placed by the algorithm while the number of checkpoints remains relatively small (e.g., less than 5 for all the platforms). This is because checkpoints are costly, and verifications help reduce the amount of time lost due to silent errors. Since verifications are cheaper, the algorithm tends to place as many of them as possible, except when their relative costs also become high (e.g., on Coastal SSD). In addition, when the number of tasks is large enough (e.g., $n > 30$ on Hera), not all tasks need to be verified. Note that there are fewer verifications for $n = 40$ than for $n = 30$, while the number of memory checkpoints remain the same. In fact, there is a threshold ($n = 38$) after which one verification for one task becomes an overkill. Instead, the

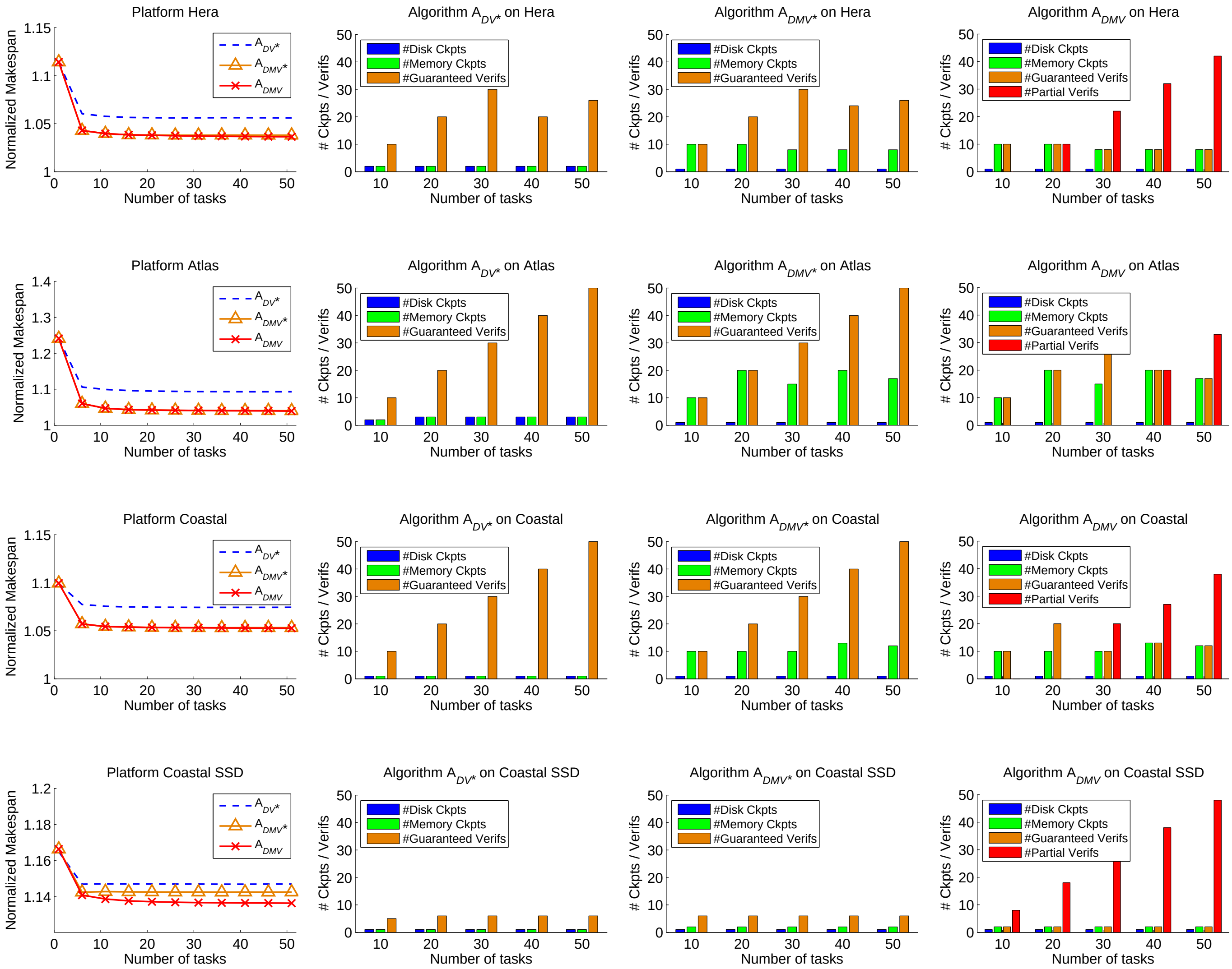


Figure 7: Performance of the three algorithms on each platform with the *Uniform* pattern. Each row corresponds to one platform.

algorithm places one verification every two tasks, resulting in exactly 20 guaranteed verifications (including the ones before each memory checkpoint) at $n = 40$ tasks.

Two-level algorithm $\mathbf{A}_{DMV^*}$. The third column of Figure 7 presents the numbers of disk checkpoints, memory checkpoints and guaranteed verifications used by the A_{DMV^*} algorithm on the four platforms and for different numbers of tasks. We observe that the number of guaranteed verifications remains similar to that placed by the A_{DV^*} algorithm. However, the two-level algorithm uses additional memory checkpoints, which drastically reduces the amount of time lost in re-execution when a silent error is detected. In particular, we observe that the algorithm A_{DMV^*} always leads to a better makespan compared to the single-level algorithm A_{DV^*}, with an improvement of 2% on Hera and 5% on Atlas, as shown in the first column of Figure 7. This demonstrates the usefulness of the multi-level checkpointing approach.

Combined algorithm $\mathbf{A}_{DMV}$. The last column of Figure 7 presents the numbers of disk checkpoints, memory checkpoints, guaranteed verifications and additional partial verifications used by the A_{DMV} algorithm on the four platforms and for different numbers of tasks. Although partial verifications are always more cost-effective than guaranteed ones, due to the imperfect recall, they are only useful if one can use a lot of them, which is only possible when the number of tasks is large enough. Therefore, the algorithm only starts to use partial verifications when the number of tasks is greater than 30 on Hera, 40 on Coastal and 50 on Atlas, where silent error rate is the highest among the four platforms. In our setting, adding partial verifications has a limited impact on the makespan, with the exception of the Coastal SSD platform, where the cost of checkpoints and verifications are much higher than on the other platforms. Partial verifications, being 100 times cheaper than guaranteed verifications, remain the only affordable resilience tool on this platform. In this case, we observe an improved makespan (around 1% with 50 tasks) compared to the A_{DMV^*} algorithm, as shown in the first column of Figure 7.

Distribution of checkpoints and verifications. Figure 8 shows the positions of the disk checkpoints, memory checkpoints, verifications and partial verifications obtained by running the A_{DMV} algorithm on each of the four platforms and for 50 tasks with the uniform distribution. For all platforms, the algorithm does not perform any additional disk checkpoints. These being costly, the algorithm rather uses more memory checkpoints and verifications. On most platforms, the optimal solution is a combination of equi-spaced memory checkpoints and guaranteed verifications, with additional partial verifications in-between. However, on the Coastal SSD platform, the cost of checkpoints and verifications is substantially higher, which leads the algorithm to choose partial verifications rather than guaranteed ones.

***Decrease* pattern.** In the following, we focus on the platforms Hera and Coastal SSD, which represent both extremes in terms of size (number of nodes) and hardware used for memory checkpointing (RAM and SSD, respectively). The first column of Figure 9 presents the performance of the three algorithms for different numbers of tasks and for the *Decrease* pattern. The second column shows the numbers of disk checkpoints, memory checkpoints, guaranteed and partial verifications given by the A_{DMV} algorithm. The third column is a visual representation of the corresponding solution obtained for 50 tasks and with the same configuration. We observe that the makespan obtained is very similar for all three algorithms (with a slight advantage for A_{DMV}). Since the large tasks at the beginning of the chain are more likely to fail, they will be checkpointed more often, as opposed to the small tasks at the end, which the algorithm does not even consider worth verifying.

***HighLow* pattern.** Once again, we focus on platforms Hera and Coastal SSD. Similarly to Figure 9, Figure 10 assesses the impact of the *HighLow* pattern on the performance of the three algorithms as well as on the numbers and the positions of checkpoints and verifications. Recall that we set the first 10% of the tasks to contain 60% of the total computational weight, while the rest of the tasks contain the remaining 40%. With 50 tasks and a total computational weight of $25000s$, the first 5 tasks have a weight of $3000s$ each, while the remaining tasks have a weight of around $222s$ each. Under this configuration, an error occurring during the execution of a large task

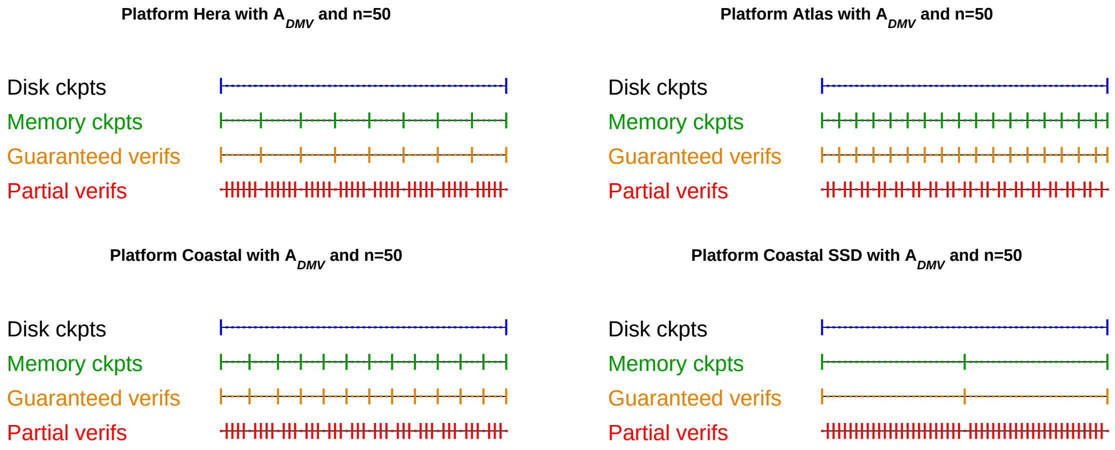


Figure 8: Distribution of disk checkpoints, memory checkpoints and verifications for the A_{DMV} algorithm on each platform with the *Uniform* pattern.

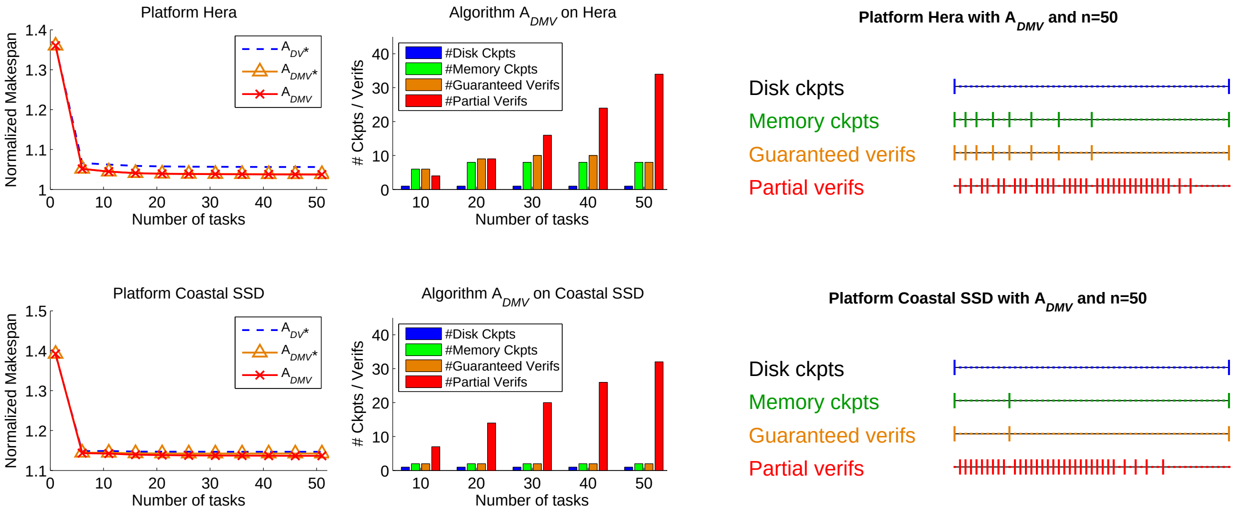


Figure 9: Performance of the three algorithms, and distribution of disk checkpoints, memory checkpoints and verifications (for the A_{DMV} algorithm) on platforms Hera and Coastal SSD with the *Decrease* pattern.

would cost $T^{\text{lost}} \approx 1500s$ time loss on average for fail-stop errors (see Equation (8)) and $3000s$ for silent errors, plus an additional $3000s$ time loss for each preceding task that has not been checkpointed. With the MTBF on Hera, a large task will fail with probability 1.3%, as opposed to the probability of 0.096% for small tasks. As a result, the disk checkpoint, which takes $300s$, turns out to be still too expensive, but the memory checkpoint, which takes only $15.4s$ on Hera, becomes mandatory: on average an error will occur way before the total accumulated cost of our preventive memory checkpoints even adds up to the cost of one task. On Coastal SSD, however, the memory checkpoint is still quite expensive, so that only one of the first 5 tasks is marked for verification and memory checkpointing. On both platforms, since the rest of the tasks are small, the solution is similar to the one we observed for the *Uniform* pattern, except that memory checkpoints and verifications are less frequent.

Summary of results. Overall, we observe that the combined use of disk checkpoints and memory checkpoints allows us to decrease the makespan, for the three task patterns and the four platforms.

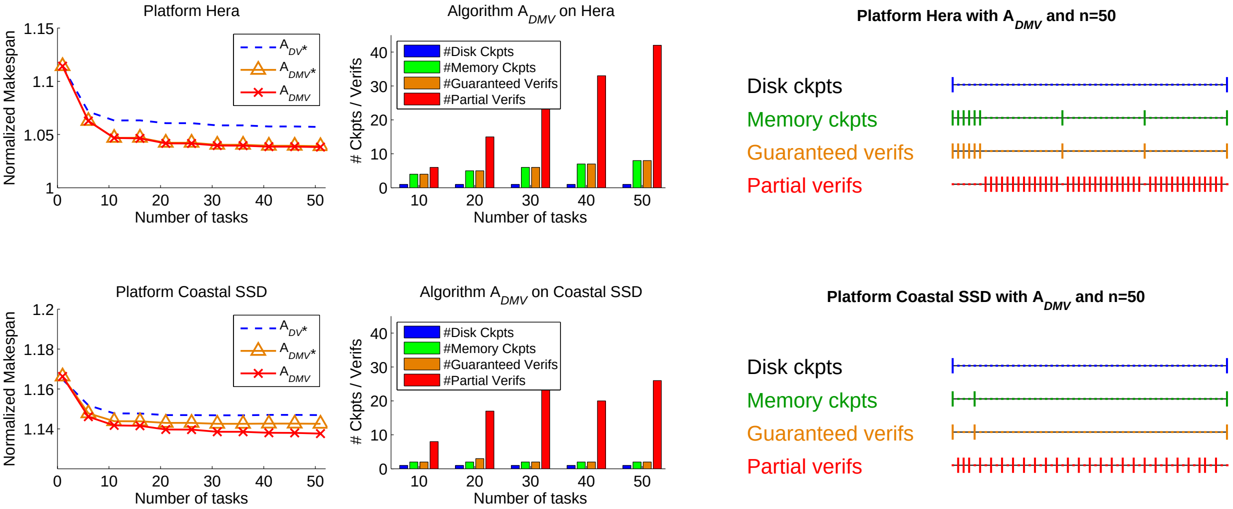


Figure 10: Performance of the three algorithms, and distribution of disk checkpoints, memory checkpoints and verifications (for the A_{DMV} algorithm) on platforms Hera and Coastal SSD with the *HighLow* pattern.

The use of partial verifications further decreases the makespan, especially on the Coastal SSD platform where the checkpointing costs are high. To give some numbers, our approach saves 2% of execution time on Hera and 5% on Atlas. These percentages may seem small, but they correspond to saving half an hour a day on Hera, and more than one hour a day on Atlas, with little further overhead.

5.2 Results for multi-level checkpointing

In this section, we perform additional experiments using a set of parameters that features $k = 3$ levels of disk checkpoints, as opposed to only one in the previous section. Therefore, we now focus on evaluating the impact of using multiple checkpointing levels to deal with fail-stop errors.

We compare three algorithms: (i) a multi-level algorithm A_{V^*} with up to $k = 3$ levels of disk checkpoints to handle both fail-stop and silent errors (with additional guaranteed verifications); and (ii) the combined algorithm A_{MV} that also uses memory checkpoints and partial verifications. Note that A_{MV} is the algorithm described in Section 4, while A_{V^*} is a simplification of this most sophisticated algorithm.

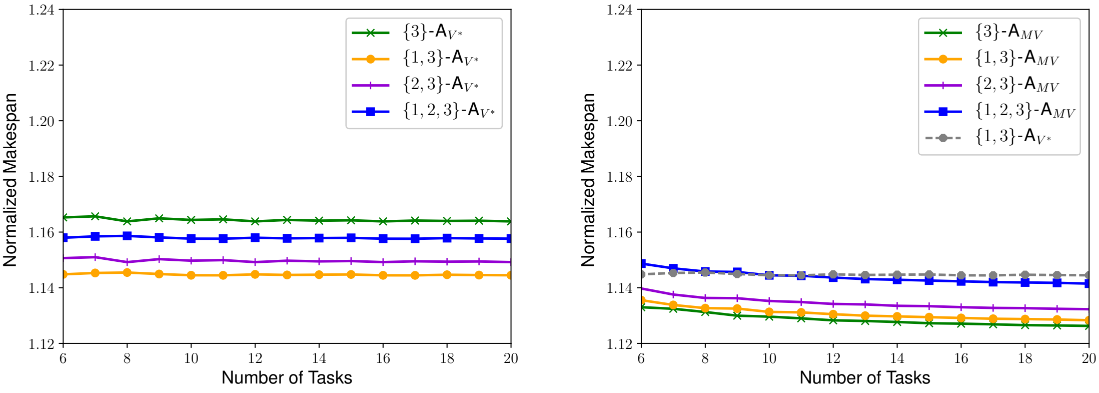
Platform settings. Table 2 presents the checkpointing costs and the associated error rates for this set of parameters, which are obtained from real measurements on the BG/Q platform Mira running LAMMPS application at ANL by Balaprakash et al. [2]. Multi-level checkpointing was provided by the FTI library [7], which offers four checkpointing levels (three levels of disk and one level of memory): local checkpoint (memory), local checkpoint + partner-copy (level-1 disk), local checkpoint + Reed-Solomon coding (level-2 disk), and PFS-based checkpoint (level-3 disk). The error rate corresponds to a default failure rate commonly used for petascale HPC applications [7, 38, 25].

Note that, with multiple levels of disk checkpoints, there is no obligation to use all available levels. In this particular case with $k = 3$ levels, one may choose among four possible subsets of levels: $\{3\}, \{1, 3\}, \{2, 3\}$, and $\{1, 2, 3\}$. Of course, we still have to account for all error types, which means that we need to adjust the error rates from the level selection as follows:

- $\{3\}$: use $\{\lambda_3 \leftarrow \lambda_1 + \lambda_2 + \lambda_3\}$;
- $\{1, 3\}$: use $\{\lambda_1\}$ and $\{\lambda_3 \leftarrow \lambda_2 + \lambda_3\}$;
- $\{2, 3\}$: use $\{\lambda_2 \leftarrow \lambda_1 + \lambda_2\}$ and $\{\lambda_3\}$;
- $\{1, 2, 3\}$: use $\{\lambda_1\}$, $\{\lambda_2\}$ and $\{\lambda_3\}$.

Set	From	Level	3	2	1	Memory
(B)	Balaprakash et al. [2]	C (s)	150	50	30	10
		λ (Hz)	1.39e-6	6.94e-6	1.39e-5	2.78e-5

Table 2: Set of parameters (B) used as input for simulations.



(a)

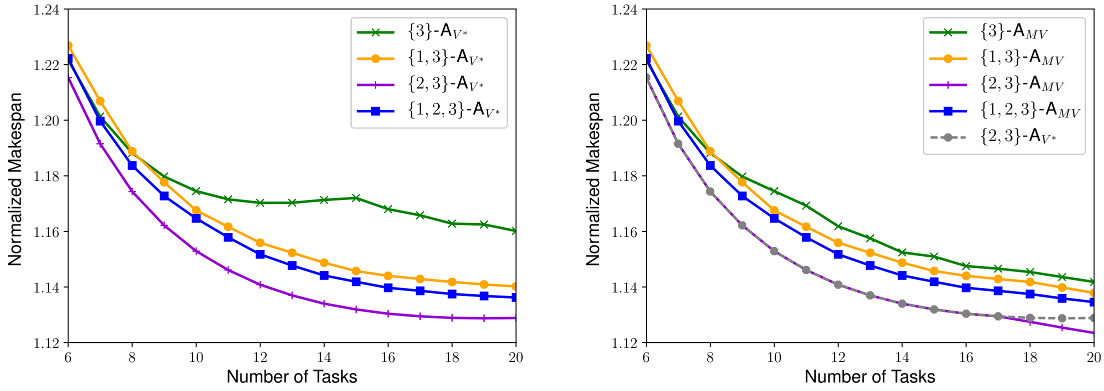
(b)

Figure 11: Performance obtained by using the optimal solution to the Multilevel-Silent problem for settings (B), using the A$_{V^*}$ algorithm (a) and the A$_{MV}$ algorithm (b), under the *Uniform* pattern with total work $W = 3600s$.

Impact of checkpointing level selection. Figure 11 presents the normalized makespan with respect to the error-free execution time obtained using the A$_{V^*}$ algorithm (a) and A$_{MV}$ algorithm (b), with up to 20 tasks under the *Uniform* pattern with total work $W = 3600s$. First, we observe that different level selections yield different overheads, but overall, using more levels does not always improve performance. In particular, we can see that, for the simple algorithm A$_{V^*}$ without additional memory checkpoints or partial verifications to deal with silent errors, the best solution is to use the $\{1, 3\}$ level selection, which achieves an overhead around 14.5%. In comparison, using only level-3 checkpoints yields an overhead around 16.5%, while using all levels $\{1, 2, 3\}$ yields an overhead just below 16%. When allowing additional memory checkpoints and partial verifications with the A$_{MV}$ algorithm, the $\{1, 3\}$ level selection no longer achieves the best results. Instead, it appears that using only level-3 checkpoints, i.e., replacing level-1 checkpoints by the cheaper memory checkpoints, yields a slightly better overhead around 13%. Overall, A$_{MV}$ improves upon the A$_{V^*}$ algorithm (with only level-3 checkpoints and guaranteed verifications) under the best level selection $\{1,\ 3\}$ by 1.5%.

Similarly to Figure 11, Figure 12 presents the normalized makespan with respect to the error-free execution time obtained using the A$_{V^*}$ algorithm (a) and A$_{MV}$ algorithm (b), with up to 20 tasks under the *Uniform* pattern with total work $W = 25000s$. Note that tasks are now significantly larger than before, and we observe that the level selection $\{2, 3\}$ beats all the other possible combinations by achieving an overhead of 13% with the A$_{V^*}$ algorithm (Figure 12a), which is further improved by another 0.5% when using the A$_{MV}$ algorithm with additional memory checkpoints for 20 tasks (Figure 12b). Since larger tasks require *more* checkpoints, but offer *limited* opportunities to achieve that goal, the algorithms tend to favor additional levels of verified checkpoints, instead of single memory checkpoints, which will be lost when a fail-stop error strikes, or single verifications (either guaranteed or partial). This is why the A$_{MV}$ algorithm becomes only slightly better with 20 tasks, and that it is not as helpful in this context.

Results for other patterns. Results for the *Decrease* and *HighLow* patterns for the combined



(a) (b)

Figure 12: Performance obtained by using the optimal solution to the Multilevel-Silent problem for settings (B), using the A_{V^*} algorithm (a) and the A_{MV} algorithm (b), under the *Uniform* pattern with total work $W = 25000s$.

problem are presented in Figures 13 and 14, respectively. As in the previous cases, we succeed to improve performance by combining the use of multi-level checkpointing and memory checkpoints (with verifications) for silent errors, especially when tasks are small (with total work $W = 3600s$). Note that Figure 14 shows a drastic drop in overhead between 10 tasks and 11 tasks. Indeed, with 10 or fewer tasks, the *HighLow* distribution consists of one big task, which has size $2160s$ when $W = 3600s$ and $15000s$ when $W = 25000s$. As a result, the probability of encountering an error (either fail-stop or silent) during the execution of the first task reaches 0.1 when $W = 3600s$ and 0.5 when $W = 25000s$, suggesting that task size plays an important role in the overhead. In comparison, with 11 or more tasks, and according to the *HighLow* distribution, we now have two big tasks instead of one. Therefore, the probability decreases to 0.05 when $W = 3600s$ and just under 1/3 when $W = 25000s$.

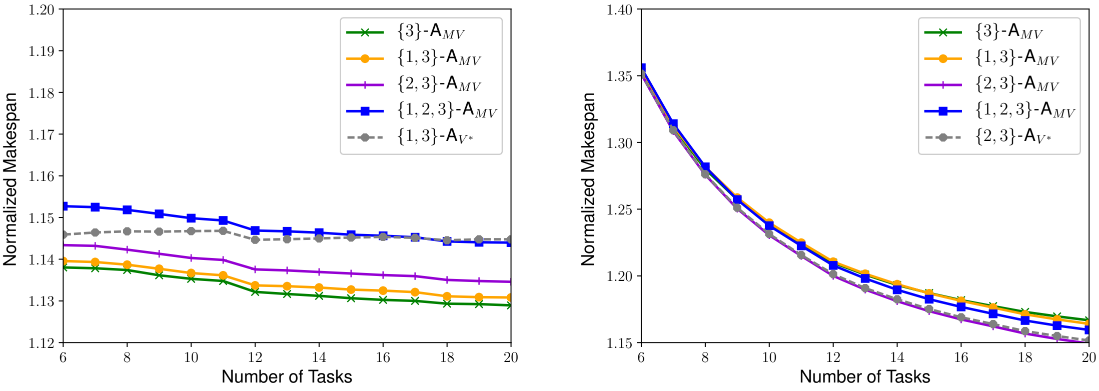
Summary of results. Overall, the simulation results have shown that the combined approach described in Section 4 to deal with both silent errors and multi-level fail-stop errors indeed leads to improved performance. In particular, when tasks are small enough, both approaches help equally to reduce the overhead. However, with fewer tasks and hence less freedom to checkpoint and verify, additional checkpoint levels seem to be favored over additional memory checkpoints or verifications. Furthermore, we have shown that the best checkpoint level selection does not always include all the levels. Finally, we remark that the implemented dynamic programming algorithms typically execute within just a few seconds and occupy up to 15GB of RAM for 20 tasks.

6 Related work

In this section, we discuss related work on fail-stop errors and silent errors, and finally outline specific results for linear workflows.

6.1 Fail-stop errors

The de-facto general-purpose error recovery technique in high performance computing is checkpoint and rollback recovery [21, 28]. For a divisible load application where checkpoints can be inserted at any point in execution for a nominal cost C, there exist well-known formulas due to Young [48] and Daly [24] to determine the optimal checkpointing period. For an application composed of a linear chain of tasks, as in this paper, the problem of finding the optimal checkpointing strategy,



(a)

(b)

Figure 13: Performance obtained by using the optimal solution to the Multilevel-Silent problem for settings (B), using the A_{V^*} algorithm under the *Decrease* pattern with total work $W = 3600s$ (a) and $W = 25000s$ (b).

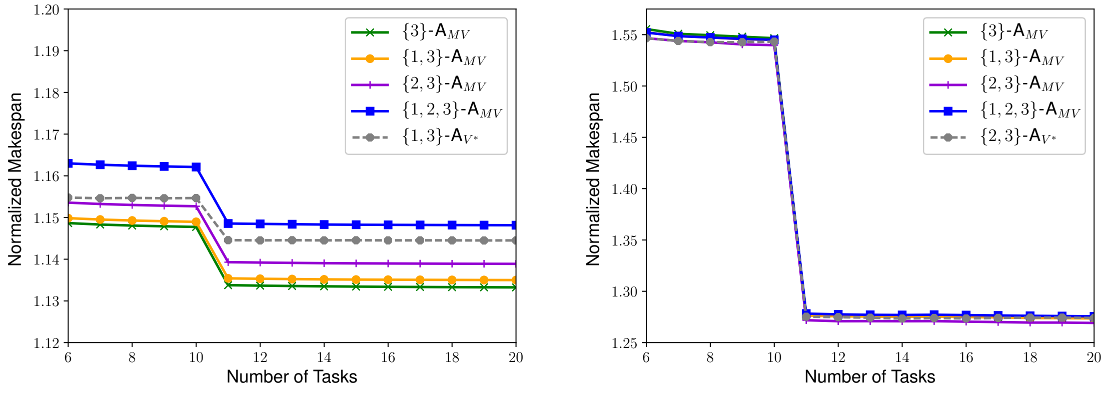
i.e., of determining which tasks to checkpoint, in order to minimize the expected execution time, has been solved by Toueg and Babaoglu [46].

However, single-level checkpointing schemes suffer from the intrinsic limitation that the cost of checkpointing/recovery grows with failure probability, and becomes unsustainable at large scale [29, 15] (even with diskless or incremental checkpointing [41]). To reduce the I/O overhead, various two-level checkpointing protocols have been studied. Vaidya [47] proposed a two-level recovery scheme that tolerates a single node failure using a local checkpoint stored on a partner node. If more than one failure occurs during any local checkpointing interval, the scheme resorts to the global checkpoint. Silva and Silva [45] advocated a similar scheme by using memory to store local checkpoints, which is protected by XOR encoding. Di et al. [26] analyzed a two-level computational pattern, and proved equal-length segments in the optimal solution. They also provided mathematical equations that can be solved numerically to compute the optimal pattern length and number of segments. Benoit et al. [10] relied on disk checkpoints to cope with fail-stop failures and used memory checkpoints coupled with error detectors to handle silent data corruptions. They derived first-order approximation formulas for the optimal pattern length as well as the number of memory checkpoints between two disk checkpoints.

Some authors have also generalized two-level checkpointing to account for an arbitrary number of levels. Moody et al. [38] implemented this approach in a three-level Scalable Checkpoint/Restart (SCR) library. They relied on a rather complex Markov model to recursively compute the efficiency of the scheme. Bautista-Gomez et al. [7] designed a four-level checkpointing library, called Fault Tolerance Interface (FTI), in which partner-copy and Reed-Solomon encoding are employed as two intermediate levels between local and global disks. Based on FTI, Di et al. [25] proposed an iterative method to compute the optimal checkpointing interval for each level with prior knowledge of the application's total execution time. Benoit [8] provided a complete characterization of multi-level checkpointing pattern based on first-order approximation, thus generalizing Young/Daly's classical results. Hakkarinen and Chen [32] considered multi-level diskless checkpointing for tolerating simultaneous failures of multiple processors. Balaprakash et al. [2] studied the trade-off between performance and energy for general multi-level checkpointing schemes.

6.2 Silent errors

Most traditional approaches maintain a single checkpoint. If the checkpoint file includes errors, the application faces an irrecoverable failure and must restart from scratch. This is because



(a)

(b)

Figure 14: Performance obtained by using the optimal solution to the Multilevel-Silent problem for settings (B), using the A_{V^*} algorithm under the *HighLow* pattern with total work $W = 3600s$ (a) and $W = 25000s$ (b).

error detection latency is ignored in traditional rollback and recovery schemes, which assume instantaneous error detection (therefore mainly targeting fail-stop failures) and are unable to accommodate silent errors. We focus in this section on related work about silent errors. A comprehensive list of techniques and references is provided by Lu, Zheng and Chien [36].

Considerable efforts have been directed at error-checking to reveal silent errors. Error detection is usually very costly. Hardware mechanisms, such as ECC memory, can detect and even correct a fraction of errors, but in practice they are complemented with software techniques. The simplest technique is triple modular redundancy and voting [37], which induces a highly costly verification. For high-performance scientific applications, process replication (each process is equipped with a replica, and messages are quadruplicated) is proposed in the RedMPI library [30]. Elliot et al. [27] combine partial redundancy and checkpointing, and confirm the benefit of dual and triple redundancy. The drawback is that twice the number of processing resources is required (for dual redundancy). An approach based on checkpointing and replication is proposed in [39], in order to detect and enable fast recovery of applications from both silent errors and hard errors.

Application-specific information can be very useful to enable ad-hoc solutions, which dramatically decrease the cost of detection. Many techniques have been advocated. They include memory scrubbing [35] and ABFT techniques [34, 16, 44], such as coding for the sparse-matrix vector multiplication kernel [44], and coupling a higher-order with a lower-order scheme for PDEs [13]. These methods can only detect an error but do not correct it. Self-stabilizing corrections after error detection in the conjugate gradient method are investigated by Sao and Vuduc [43]. Heroux and Hoemmen [33] design a fault-tolerant GMRES capable of converging despite silent errors. Bronevetsky and de Supinski [17] provide a comparative study of detection costs for iterative methods.

Recently, detectors based on data analytics have been proposed to serve as partial verifications [14, 5, 6]. These detectors use interpolation techniques, such as time series prediction and spatial multivariate interpolation, on scientific dataset to offer large error coverage for a negligible overhead. Although not perfect, their accuracy-to-cost ratios tend to be very high, which makes them interesting alternatives at large scale. For divisible load applications, periodic patterns with partial and guaranteed verifications are studied in [10]. We point out that the approach described in this paper is agnostic of the underlying error-detection technique and takes the cost of verification as an input parameter to the model.

6.3 Linear workflows

In this section, we focus on work related to linear workflows. The main difference with divisible load applications is that one can insert resilience mechanisms only at the end of the execution of a task. We may well have a limited number of tasks, which prevents the use of any periodic strategy à la Young/Daly [48, 24]. Instead, the optimal solution for any linear task graph is typically obtained with dynamic programming algorithms.

As already mentioned, Toueg and Babaoglu [46] have dealt with single-level checkpointing for fail-stop errors. Their work has been extended in [9] to deal with silent errors in addition to fail-stop errors. The approach in [9] uses only guaranteed verifications and one-level of checkpointing. It has been further extended in [11] (the preliminary version of this paper) to include partial verifications in addition to guaranteed verification, and in-memory checkpointing in addition to disk checkpointing.

This work provides the last step and shows how to add multi-level disk checkpointing protocols. We now deal with k disk checkpoint levels (where k is arbitrary), one memory checkpoint level, and partial and guaranteed verifications. As a result, we combine the most efficient techniques for fail-stop and silent errors within a unified framework.

7 Conclusion

In this paper, we focused on HPC applications whose dependency graph forms a linear chain, and we proposed two important extensions to single-level checkpointing, allowing us to cope with both multi-level fail-stop errors and silent data corruptions, on large-scale platforms. Although numerous studies have dealt with either error source, few studies have dealt with both, while it is mandatory to address both sources simultaneously at scale. We have combined the multi-level disk checkpointing technique with in-memory checkpoints and verification mechanisms (partial or guaranteed), and we have designed a sophisticated multi-level dynamic programming algorithm that computes the optimal solution for a linear application workflow in polynomial time.

Simulations based on realistic parameters on several platforms show consistent results, and confirm the benefit of the combined approach. Improvement can be seen both by using additional guaranteed and/or partial verifications for silent errors, and by selecting several levels of checkpoints, between those offered by the platform, to handle different types of fail-stop errors. While the most general algorithm has a high complexity of $O(n^{k+5})$, where n is the number of tasks and k is the number of checkpointing levels, it executes within a few seconds for $n = 20$ tasks and $k = 3$ levels, and therefore can be readily used for real-life linear workflows whose sizes rarely exceed tens of tasks.

One interesting future direction is to assess the usefulness of this approach on general application workflows. The problem gets much more challenging, even in the simplified scenario where each task requires the entire platform to execute. In fact, in this simplified scenario, it is already NP-hard to decide which task to checkpoint in a simple join graph ($n-1$ source tasks and a common sink task), with only fail-stop errors striking (hence a single level of checkpoint and no verification at all) [1]. Still, heuristics are urgently needed to address the same problem as in this paper, with several error sources, several checkpoint types, and two verification mechanisms, if we are to deploy general HPC workflows efficiently at scale.

Acknowledgments

This research was funded in part by the European project SCoRPiO, by the LABEX MILYON (ANR-10-LABX-0070) of Université de Lyon, within the program "Investissements d'Avenir" (ANR-11-IDEX-0007) operated by the French National Research Agency (ANR), and by the PIA ELCI project. Yves Robert is with Institut Universitaire de France.

References

[1] G. Aupy, A. Benoit, H. Casanova, and Y. Robert. Scheduling computational workflows on failure-prone platforms. In *17th Workshop on Advances in Parallel and Distributed Computational Models (APDCM'15)*, 2015.

[2] P. Balaprakash, L. A. B. Gomez, M.-S. Bouguerra, S. M. Wild, F. Cappello, and P. D. Hovland. Analysis of the tradeoffs between energy and run time for multilevel checkpointing. In *Proc. PMBS'14*, 2014.

[3] L. Bautista-Gomez, A. Benoit, A. Cavelan, S. K. Raina, Y. Robert, and H. Sun. Which verification for soft error detection? In *Proc. HiPC'15*, 2015.

[4] L. Bautista-Gomez, A. Benoit, A. Cavelan, S. K. Raina, Y. Robert, and H. Sun. Coping with recall and precision of soft error detectors. *Journal of Parallel and Distributed Computing*, 98:8–24, 2016.

[5] L. Bautista Gomez and F. Cappello. Detecting silent data corruption through data dynamic monitoring for scientific applications. *SIGPLAN Notices*, 49(8):381–382, 2014.

[6] L. Bautista Gomez and F. Cappello. Detecting and correcting data corruption in stencil applications through multivariate interpolation. In *Proc.1st Int. Workshop on Fault Tolerant Systems (FTS)*, 2015.

[7] L. Bautista-Gomez, S. Tsuboi, D. Komatitsch, F. Cappello, N. Maruyama, and S. Matsuoka. FTI: High performance fault tolerance interface for hybrid systems. In *Proc. SC'11*, 2011.

[8] A. Benoit, A. Cavelan, V. Le Fèvre, Y. Robert, and H. Sun. Towards optimal multi-level checkpointing. *IEEE Transactions on Computers*, 2017.

[9] A. Benoit, A. Cavelan, Y. Robert, and H. Sun. Assessing general-purpose algorithms to cope with fail-stop and silent errors. *ACM Trans. Parallel Computing*, 3(2), 2016.

[10] A. Benoit, A. Cavelan, Y. Robert, and H. Sun. Optimal resilience patterns to cope with fail-stop and silent errors. In *Proc. IPDPS'16*, 2016. Full version available as INRIA RR-8786.

[11] A. Benoit, A. Cavelan, Y. Robert, and H. Sun. Two-level checkpointing and partial verifications for linear task graphs. In *PDSEC'2016, the 17th Workshop on Parallel and Distributed Scientific and Engineering Computing*. IEEE Computer Society Press, 2016.

[12] A. Benoit, Y. Robert, and S. K. Raina. Efficient checkpoint/verification patterns. *Int. J. High Performance Computing Applications*, 2015.

[13] A. R. Benson, S. Schmit, and R. Schreiber. Silent error detection in numerical time-stepping schemes. *Int. J. High Performance Computing Applications*, 2014.

[14] E. Berrocal, L. Bautista-Gomez, S. Di, Z. Lan, and F. Cappello. Lightweight silent data corruption detection based on runtime data analysis for HPC applications. In *Proc. HPDC*, 2015.

[15] G. Bosilca, A. Bouteiller, E. Brunet, F. Cappello, J. Dongarra, A. Guermouche, T. Herault, Y. Robert, F. Vivien, and D. Zaidouni. Unified model for assessing checkpointing protocols at extreme-scale. *Concurrency and Computation: Practice and Experience*, 2013.

[16] G. Bosilca, R. Delmas, J. Dongarra, and J. Langou. Algorithm-based fault tolerance applied to high performance computing. *J. Parallel Distrib. Comput.*, 69(4):410–416, 2009.

[17] G. Bronevetsky and B. de Supinski. Soft error vulnerability of iterative linear algebra methods. In *Proceedings of the International Conference on Supercomputing (ICS)*, pages 155–164, 2008.

[18] F. Cappello, A. Geist, B. Gropp, L. Kale, B. Kramer, and M. Snir. Toward Exascale Resilience. *Int. Journal of High Performance Computing Applications*, 23(4):374–388, 2009.

[19] F. Cappello, A. Geist, W. Gropp, S. Kale, B. Kramer, and M. Snir. Toward exascale resilience: 2014 update. *Supercomputing frontiers and innovations*, 1(1), 2014.

[20] A. Cavelan, S. K. Raina, Y. Robert, and H. Sun. Assessing the impact of partial verifications against silent data corruptions. In *Proc. ICPP*, 2015.

[21] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, 1985.

[22] Z. Chen. Online-ABFT: An online algorithm based fault tolerance scheme for soft error detection in iterative methods. In *Proc. PPoPP*, pages 167–176, 2013.

[23] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms.* The MIT Press, 2001.

[24] J. T. Daly. A higher order estimate of the optimum checkpoint interval for restart dumps. *Future Generation Comp. Syst.*, 22(3):303–312, 2006.

[25] S. Di, M. S. Bouguerra, L. Bautista-Gomez, and F. Cappello. Optimization of multi-level checkpoint model for large scale HPC applications. In *Proc. IPDPS'14*, 2014.

[26] S. Di, Y. Robert, F. Vivien, and F. Cappello. Toward an optimal online checkpoint solution under a two-level HPC checkpoint model. *IEEE Trans. Parallel & Distributed Systems*, 2016, preprint available on the IEEE digital library.

[27] J. Elliott, K. Kharbas, D. Fiala, F. Mueller, K. Ferreira, and C. Engelmann. Combining partial redundancy and checkpointing for HPC. In *Proceedings of the IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 615–626, 2012.

[28] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Survey*, 34:375–408, 2002.

[29] K. Ferreira, J. Stearley, J. H. I. Laros, R. Oldfield, K. Pedretti, R. Brightwell, R. Riesen, P. G. Bridges, and D. Arnold. Evaluating the Viability of Process Replication Reliability for Exascale Systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 44:1–44:12, 2011.

[30] D. Fiala, F. Mueller, C. Engelmann, R. Riesen, K. Ferreira, and R. Brightwell. Detection and correction of silent data corruption for large-scale high-performance computing. In *Proc. SC'12*, page 78, 2012.

[31] R. G. Gallager. *Stochastic Processes: Theory for Applications.* Cambridge University Press, New York, NY, USA, 2014.

[32] D. Hakkarinen and Z. Chen. Multilevel diskless checkpointing. *IEEE Transactions on Computers*, 62(4):772–783, 2013.

[33] M. Heroux and M. Hoemmen. Fault-tolerant iterative methods via selective reliability. Research report SAND2011-3915 C, Sandia National Laboratories, 2011.

[34] K.-H. Huang and J. A. Abraham. Algorithm-based fault tolerance for matrix operations. *IEEE Trans. Comput.*, 33(6):518–528, 1984.

[35] A. A. Hwang, I. A. Stefanovici, and B. Schroeder. Cosmic rays don't strike twice: understanding the nature of DRAM errors and the implications for system design. *SIGARCH Comput. Archit. News*, 40(1):111–122, 2012.

[36] G. Lu, Z. Zheng, and A. A. Chien. When is multi-version checkpointing needed? In *Proc. 3rd Workshop on Fault-tolerance for HPC at extreme scale (FTXS)*, pages 49–56, 2013.

[37] R. E. Lyons and W. Vanderkulk. The use of triple-modular redundancy to improve computer reliability. *IBM J. Res. Dev.*, 6(2):200–209, 1962.

[38] A. Moody, G. Bronevetsky, K. Mohror, and B. R. d. Supinski. Design, modeling, and evaluation of a scalable multi-level checkpointing system. In *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC'10)*, pages 1–11, 2010.

[39] X. Ni, E. Meneses, N. Jain, and L. V. Kalé. ACR: Automatic Checkpoint/Restart for Soft and Hard Error Protection. In *Proc. SC'13*. ACM, 2013.

[40] T. O'Gorman. The effect of cosmic rays on the soft error rate of a DRAM at ground level. *IEEE Trans. Electron Devices*, 41(4):553–557, 1994.

[41] J. Plank, K. Li, and M. Puening. Diskless checkpointing. *IEEE Trans. Parallel Dist. Systems*, 9(10):972–986, 1998.

[42] F. Quaglia. A cost model for selecting checkpoint positions in time warp parallel simulation. *IEEE Trans. Parallel Dist. Syst.*, 12(4):346–362, 2001.

[43] P. Sao and R. Vuduc. Self-stabilizing iterative solvers. In *Proc. ScalA '13*, 2013.

[44] M. Shantharam, S. Srinivasmurthy, and P. Raghavan. Fault tolerant preconditioned conjugate gradient for sparse linear system solution. In *Proc. ICS*, pages 69–78, 2012.

[45] L. Silva and J. Silva. Using two-level stable storage for efficient checkpointing. *IEE Proceedings - Software*, 145(6):198–202, 1998.

[46] S. Toueg and Ö. Babaoglu. On the optimum checkpoint selection problem. *SIAM J. Comput.*, 13(3):630–649, 1984.

[47] N. H. Vaidya. A case for two-level distributed recovery schemes. *SIGMETRICS Perform. Eval. Rev.*, 23(1):64–73, 1995.

[48] J. W. Young. A first order approximation to the optimum checkpoint interval. *Comm. of the ACM*, 17(9):530–531, 1974.

[49] J. Ziegler, M. Nelson, J. Shell, R. Peterson, C. Gelderloos, H. Muhlfeld, and C. Montrose. Cosmic ray soft error rates of 16-Mb DRAM memory chips. *IEEE Journal of Solid-State Circuits*, 33(2):246–252, 1998.

[50] J. F. Ziegler, H. W. Curtis, H. P. Muhlfeld, C. J. Montrose, and B. Chin. IBM experiments in soft fails in computer electronics. *IBM J. Res. Dev.*, 40(1):3–18, 1996.

RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l’Europe Montbonnot
38334 Saint Ismier Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399